ON THE PROPERTIES AND APPLICATIONS OF PROGRAM SCHEMAS

Ashok K. Chandra

Stanford University

Prepared for:

Advanced Research Projects Agency

March 1973

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-188

STAN-CS-73-336

ON THE PROPERTIES AND APPLICATIONS OF PROGRAM SCHEMAS

BY

ASHOK K. CHANDRA

SUPPORTED BY
ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 457

MARCH 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-188

MARCH 1973

COMPUTER SCIENCE DEPARTMENT
REPORT STAN-CS-73-336

ON THE PROPERTIES AND APPLICATIONS OF PROGRAM SCHEMAS

by

Ashok K. Chandra

ABSTRACT: The interesting questions one can ask about program schemas include questions about the "power" of classes of shemas and their decision problems viz. halting, divergence, equivalence, etc. We first consider the powers of schemas with various features: recursion, equality tests, and several data structures such as pushdown stacks, lists, queues and arrays. We then consider the decision problems for schemas with equality and with commutative and invertible functions. Finally a generalized class of schemas is described in an attempt to unify the various classes of uninterpreted and semi-interpreted schemas and schemas with special data structures.

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract No. SD-183.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

Reproduced in the USA. Available from the National Technical Information Service, Springfield, Virginia 22151.

Acknowledgment

I would like to express my sincere gratitude to my advisor, Professor Zohar Manna, for his stimulating suggestions and his constant guidance and encouragement during the course of this research. I also take this opportunity to thank my Professors, in particular, Robert Floyd and Donald Knuth, for teaching me what research is all about. My fellow students contributed to this work through their helpful discussions, and Phyllis Winkler did so in a more tangible fashion by her excellent typing and her unwavering belief that my iterations on the manuscript would eventually converge.

I dedicate this work to my parents.

Table of Contents

Chapter 1. Introduction

Program Schemas and Their Applications

A program schema is a computer program in which the basic functions and predicates are left unspecified. Essentially, a program schema depicts the control structure of the program, and leaves most of the details to be specified in an interpretation for the functions and predicates of the schema. Thus, a schema is not encumbered with the details of the actual domain of the values it computes on. This basic approach can be used to develop a machine-independent theory of computation. Of course, it is not intended that such a theory will replace the other approaches that have proved useful, such as recursive function theory, complexity theory, automata theory, the fixpoint theory of computation and Scott's lattice-theory approach to computation. Instead, it is expected to supplement these by providing a model for computation in which certain useful facts can be expressed, clarified, and understood.

Some of the applications of schemata theory are the following.

1. Comparing the power of programming features. By "power" we mean the ability to program in a "natural" way. Interpreted programs are not very useful for comparing power because interpreted programming languages are caught very easily in the mire of Turing machine computability. For example, iterative programs with just three counters can compute any "computable" function. Yet, all programmers are aware that recursion is more "powerful" than iteration alone, and that a pushdown

stack can be used to eliminate recursion. These notions become transparent at the level of schemas. It is not expected, of course, that schemas will give a complete characterization of the intuitive notion of power since even informally there does not seem to be complete agreement on this notion. But it is hoped that schemas will give an approximation one step better than interpreted programs, and possibly lead the way for further studies.

2. Another application of schemata theory is in the study of program optimization. This is to be expected because optimization often involves changing the control structure of a program without altering the outcome of the computation. Closely related to the question of program optimization is the problem of recursion removal. To give an example, consider the recursive program

$$F(y) \Leftarrow \underline{\text{if}}\ p(y)\ \underline{\text{then}}\ a\ \underline{\text{else}}\ F(f(y))$$

where p represents some predicate test, f represents some function, and a is some constant. It is clear that the recursive call $F(f(y))$ can be replaced by iteration: change the value of the variable y to $f(y)$ and repeat the "$\underline{\text{if}}\ p(y)\ \underline{\text{then}}\ \ldots$" statement. In fact, this kind of an optimization has been introduced in many compilers. Now, consider the following program

$$F(y) \Leftarrow \underline{\text{if}}\ p(y)\ \underline{\text{then}}\ a\ \underline{\text{else}}\ g(y,F(f(y)))\ .$$

Can this recursion be replaced by iteration? The answer is yes, though in general the iterative program takes more time than the recursive program. Sometimes, however, we can make use of particular properties of the functions f and g to obtain more efficient code. For example,

if the function g is associative, this fact can be used to transform the recursive program into one that is essentially iterative (analogous to the earlier example):

$$F(y) \Leftarrow \underline{\text{if}}\ p(y)\ \underline{\text{then}}\ a\ \underline{\text{else}}\ G(y,f(y))$$

$$G(x,y) \Leftarrow \underline{\text{if}}\ p(y)\ \underline{\text{then}}\ g(x,a)\ \underline{\text{else}}\ G(g(x,y),f(y)) \quad .$$

This example points out a limitation of the assumption that all base functions and predicates be completely uninterpreted, because if such an assumption is strictly adhered to, then the translation described above is not valid because it assumes the associativity of the function g . What has happened is that by an insistance on modeling only the control structure of our program (by saying that all base functions and predicates must be uninterpreted) we have obtained a model that fails to embody the same essential relations on the domain of the program we were trying to model. It seems, therefore, that in order to have a useful theory of computation we must back off from a rigid stance of completely uninterpreted base functions and predicates, and should allow semi-interpreted schemas in the theory.

3. A third application of schemata theory is proving properties about deterministic processes (by "deterministic" we mean deterministic as against intuitive, and not as against stochastic, or nondeterministic as in automata theory). For our purposes computer programs are the most important of the deterministic processes (readers who have spent long hours trying to debug programs might object to the use of the word "deterministic" as applied to computer programs -- nevertheless, we

persist). Another example of a deterministic process is a finite automaton. A side effect of proving properties about schemas, and one that has received scant attention to date, is that once certain properties are proved about schemas they apply to all the processes that are modeled by the schema (see Chandra [1972]). In this way several results can be proven simultaneously simply by proving the corresponding result for an appropriate schema; and conceivably, schemas could also be used to interrelate various results in different fields of the theory of computation.

To give an example, the equivalence of two programs can be proven, in many cases, by proving the equivalence of the corresponding schemas. Frequently, however, we need some additional information about the interrelations between the base functions. Consider the following two programs on natural numbers, where x and y are the inputs, and z is the output.

(1) $z \leftarrow x*y$

(2) $x_1 \leftarrow 0$; $y_1 \leftarrow y$;
<u>while</u> $y_1 \neq 0$ <u>do</u> <u>begin</u> $x_1 \leftarrow x+x_1$; $y_1 \leftarrow y_1-1$ <u>end</u>;
$z \leftarrow x_1$.

We certainly cannot prove the equivalence of these two programs by replacing the various functions (multiplication, addition, subtraction) by uninterpreted functions. Instead, we need the property that multiplication is related to addition in a certain way, in fact, multiplication is defined by the function F in (3) below. Using this additional piece of information we can prove the equivalence of (1) and (2) as follows.

(3) $F(x,y) \Leftarrow$ **if** $y = 0$ **then** 0 **else** $x+F(x,y-1)$

(4) $F(x,y) \Leftarrow$ **if** $p(y)$ **then** a **else** $g(x,F(x,f(y)))$

(5) $F(x,y) \Leftarrow G(x,y,a,y)$

$G(x,y,x_1,y_1) \Leftarrow$ **if** $p(y_1)$ **then** x_1 **else** $G(x,y,g(x,x_1),f(y_1))$

(6) $x_1 \leftarrow a$; $y_1 \leftarrow y$;

while $\neg\, p(y_1)$ **do begin** $x_1 \leftarrow g(x,x_1)$; $y_1 \leftarrow f(y_1)$ **end**

$z \leftarrow x_1$.

We replace (3) by its corresponding schema (4), translate it to an equivalent schema (5) and finally change the form to make it purely iterative (6). Now, in this schema, if we substitute the meanings of the base functions and predicates we have precisely the desired program (2). One might well ask why we used schemas in this example. The reason is that this clearly separates the semantic part of the procedure from the syntactic part since the steps (4) to (5), and (5) to (6) were purely a matter of symbol manipulation. But there is a very desirable side effect of this method. Having proved the equivalence of (4) and (6) once, we can also use it to prove the equivalence of the programs (7) and (8) where the operation of exponentiation (x^y) is defined by the function F in (9).

(7) $z \leftarrow x^y$

(8) $x_1 \leftarrow 1$; $y_1 \leftarrow y$;

while $y_1 \neq 0$ **do begin** $x_1 \leftarrow x*x_1$; $y_1 \leftarrow y_1-1$ **end**;

$z \leftarrow x_1$

(9) $F(x,y) \Leftarrow$ **if** $y = 0$ **then** 1 **else** $x*F(x,y-1)$

We should state that the preceding is merely an intuitive elaboration rather than any attempt at a formal presentation of what schemas can be useful for.

Historical Remarks

The study of program schemas can be traced back to the work of Ianov [1958, 1960] where he treated the entire data space of a program as being representable by a single value which could be changed by applying functions, or tested by applying predicates to it. These base functions and predicates were assumed to be total, but otherwise uninterpreted. This model of computation is quite closely related to finite state machines, and, as may be expected, the problems of termination and equivalence of Ianov schemas are decidable. In this regard, the work of Rutledge [1964] is also to be noted.

But this simple model of computation is not adequate for describing most computations. To obtain a better description we would require that the functions and predicates of the schema be related in some way. For example, the data space in real computations is usually divided into individual components, and functions and predicates are applied to these components. A convenient way of handling the subdivision of memory (Paterson [1967, 1968], Luckham, Park and Paterson [1970]) is to consider schemas containing several variables (also called registers), one for each component of the data space. The base functions and predicates are left uninterpreted. We argue in Section 4.1, however, that these basic concepts, viz., the explicit subdivision of data space and the use of uninterpreted base functions and predicates, are not as general as could be desired, and we attempt to remedy this situation.

Subsequent work in schemata theory has been in studying the effects of additional features, for example, the use of recursion, counters, pushdown stacks, arrays, parallel computations, partial functions in the interpretations, etc. Without attempting a complete list of contributions, we note the important works of Karp and Miller [1969], Paterson and Hewitt [1971], Strong [1971], Garland and Luckham [1971], and Constable and Gries [1972]. It is interesting to note that the earlier works tend to focus on the decision problems of schemas, namely, the halting, divergence and equivalence problems for schemas, and subsequent works mainly deal with the problems of translation from one class of schemas to another class.

Outline of the Thesis

In this thesis we restrict our attention to schemas with no explicit inputs: zero-ary functions (individual constants) serve the role of inputs. The interpretations for a schema describe total functions and predicates over arbitrary domains -- we do not allow partial functions or predicates in an interpretation.

The chapters have been organized so as to separate the main results and the intuitive discussion from the detailed proofs and examples which relatively few readers would like to plow through anyway. Most of the material requires no prior knowledge of schemas, but many of the proofs assume a familiarity with the basic methods used by other researchers.

Most of the notation and introductory material on schemas is contained in Section 2.1. Section 2.2 discusses a relation between schemas and formal languages via value languages of schemas. This leads up to a discussion

on the power of various classes of schemas in Section 2.3. Chapter 3 deals with the decision problems of schemas. The first part (Section 3.1) considers uninterpreted flowchart schemas with equality tests. The second part (Section 3.2) considers semi-interpreted schemas, and, in particular, considers the effect of commutativity and invertibility relations on the decision problems. The final chapter, Chapter 4, introduces a class of generalized schemas. The formalism of a first order theory is used to unify the data structures used by schemas with the base values on which the schemas compute, and it is shown that much of conventional schemata theory can be represented within this framework.

Chapter 2. Translation Problems

2.1 Introduction

In this section we introduce the basic definitions and terminology to be used in later sections. Only the simplest of proofs are given in the main exposition, the others are postponed to Section 2.1.9.

In the development of many theories (e.g. number theory) it has turned out that the most fundamental questions (e.g. what is a natural number) are answered quite late in the development. Part of the reason for this is that the answers to these questions are unnecessary for an intuitive understanding of much of the theory, and the formalism necessary to answer them can detract from the simplicity of the rest of the theory. In accordance with this view, we will be quite informal on many points, namely, on the following questions:

(a) what is a schema,

(b) what is an interpretation corresponding to a schema,

(c) what is an uninterpreted schema,

(d) what does the "value of a variable" mean.

The answers to these questions are obvious for the schemas we present in this chapter and in the next one, and we dispense with formalities until the last chapter which defines a formal notion of schemas.

2.1.1 Flowchart Schemas

A flowchart schema S has a finite number of variables represented by the symbols $y_1, y_2, \ldots, y_n$, $z_1, z_2, \ldots, z_m$. It uses uninterpreted functions $f_1, f_2, \ldots, f_r$ and predicates $p_1, p_2, \ldots, p_s$ called the base

functions and base predicates. We should caution the reader at this point that we will not restrict ourselves to the use of just these symbols to denote variables, functions and predicates when convenience and clarity demand otherwise. Some of the base functions may be zero-ary functions, also called individual constants, and usually denoted by the symbols $a_1, a_2, \ldots$. A term τ can be built up using the variables $y_1, \ldots, y_n$ of the schema and the zero-ary functions, and applying the other functions to them. We use the notation $\tau(y_{i_1}, y_{i_2}, \ldots)$ to indicate that no variables other than $y_{i_1}, y_{i_2}, \ldots$ appear in the term τ , for example, $\tau(y_1, y_3)$ indicates that no variable other than y_1 and y_3 appears in τ , but it is not necessary that both have to appear. In accordance with this nomenclature, $\tau()$ denotes a constant term, that is, a term that has no variables in it. A monadic schema is a schema in which only zero-ary and unary functions and predicates are used.

An interpretation I over a domain D contains the functions and predicates $f_1^I, \ldots, f_r^I, p_1^I, \ldots, p_s^I$ which correspond to the function and predicate symbols $f_1, \ldots, f_r, p_1, \ldots, p_s$ of a schema. If f_i is a k-ary function symbol, then $f_i^I : D^k \rightarrow D$; likewise, if p_i is a k-ary predicate symbol, then $p_i^I : D^k \rightarrow B$ where B is the boolean domain {true, false} . We will usually not distinguish between the symbols f_i and f_i^I , and we will write the latter simply as f_i , with the interpretation I being understood.

A schema is said to be uninterpreted if all interpretations which specify (at least) all the base functions and predicates of the schema, are allowed. A schema is said to be interpreted (partially interpreted) if not all

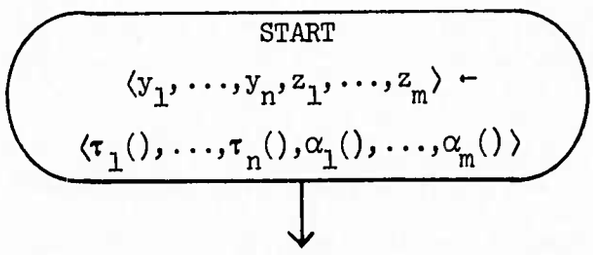
interpretations are allowed. If I is an interpretation that is allowed for S, we say I is an interpretation for S, and S admits I.

It is clear that a schema uses two kinds of values -- base values which are elements of the domain D of the interpretation, and boolean values, which are elements of the domain B. Now the mystery of two kinds of variables y_i and z_i can be clarified. The variables of the form y_i take on base values, and variables z_i take boolean values. The y_i's are called data variables, or just variables for short; the z_i's are called boolean variables.

An atomic formula is a boolean value, a boolean variable, or $p(\tau_1,\ldots,\tau_k)$ where p is a k-ary predicate. We use the symbol α to denote an atomic formula or a negated atomic formula -- sometimes called a primitive formula. In accordance with the nomenclature for terms, $\alpha()$ indicates a constant atomic formula (or negated atomic formula).

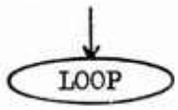
The statements of a flowchart schema are of the following types (there is a single start statement in the schema):

Start statement:

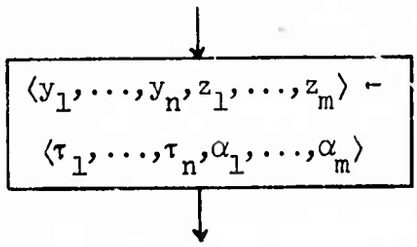


Halt statement:

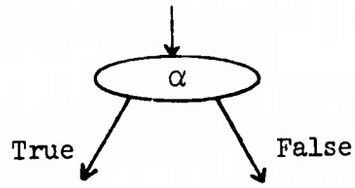
Loop statement:



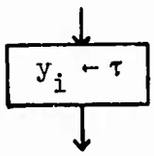
Assignment statement:

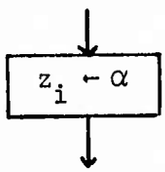


Test statement:



The assignment statement simultaneously reassigns the values of all variables. Often, only a few of these are to be changed, and for convenience, we allow the abbreviations





which indicate that all variables not explicitly mentioned are unchanged.

To represent flowchart schemas we will usually use the more compact ALGOL-like notation, allowing the use of labels $(L_1, L_2, \ldots)$ and goto

statements. In addition, we also allow the use of block structure, if-then-else statements, while statements, and nonrecursive procedures with the tacit understanding that these features can be eliminated, using goto statements instead to get a "legal" flowchart schema.

Ianov schemas are about the simplest kinds of flowchart schemas. A Ianov schema has a single variable y, and its statements are of the following types:

(1) START $y \leftarrow a$,

(2) HALT(y) ,

(3) LOOP ,

(4) $y \leftarrow f_i(y)$, and

(5) if $p_1(y)$ then goto L_1 else goto L_2 .

A flowchart schema with equality is a flowchart schema with the addition that atomic formulas of the form

$$\tau_1 = \tau_2$$

are also allowed.

Currently there seems to be a little misunderstanding as to the role of schemas with equality. In our treatment a flowchart schema with equality is not a partially interpreted schema because absolutely no restriction is placed on the interpretations allowed. This point is considered in greater detail in Section 2.1.9.

The class of flowchart schemas will be denoted by $\mathcal{C}()$, and flowchart schemas with equality by $\mathcal{C}(=)$. The class of flowchart schemas that use no more than n data variables is $\mathcal{C}(n \text{ var})$, and similarly $\mathcal{C}(n \text{ var}, =)$ for equality schemas. Note: schemas in $\mathcal{C}(n \text{ var})$ or in

$\mathcal{C}$(n var, =) may have an arbitrary number of boolean variables.

2.1.2 Augmented Schemas

We will also consider flowchart schemas augmented with (structural) features designed to make the schemas more powerful.

A counter is a variable (usually denoted by the letter c) whose values are non-negative integers. All counters used by a schema are initialized to zero by the start statement. The operations allowed on a counter are

(1) $c \leftarrow c+1$,

(2) $c \leftarrow c-1$, and

(3) if $c = 0$ then goto L_1 else goto L_2 ,

where L_1, L_2 are arbitrary labels. The subtraction (diminish) operator in $c \leftarrow c-1$ is on natural numbers, that is, $0-1 = 0$. The class of schemas with counters is designated $\mathcal{C}$(c) , schemas with at most one counter $\mathcal{C}$(1c) , with a counter and equality $\mathcal{C}$(1c,=) , and so on in the obvious way.

A pushdown stack (usually denoted by the symbol s) is a last-in first-out store which can hold values of both types (data, and boolean). A schema with a stack can "push" a data value and a boolean value into the stack, it can "pop" them from the "top", and it can test to see if the stack is empty. The statements allowed are:

(1) $s \leftarrow push(s,y,z)$, and

(2) if $s = \Lambda$ then goto L else begin $\langle y,z \rangle \leftarrow top(s)$; $s \leftarrow pop(s)$ end ,

where y denotes an arbitrary data variable, z a boolean variable, Λ the empty stack, and L a label. The start statement in a schema initializes all stacks to be empty. The class of schemas with pushdown stacks is $\mathcal{C}(pds)$, with at most one stack $\mathcal{C}(lpds)$, etc.

A queue (usually denoted by q) is a first-in first-out store. A schema with a queue can "add" values at one end, and "remove" them from the other end (first(q)), and it can test to see if the queue is empty. The statements for a queue are:

(1) $q \leftarrow add(q,y,z)$, and

(2) if $q = \Lambda$ then goto L else begin $\langle y,z \rangle \leftarrow first(q)$; $q \leftarrow remove(q)$ end .

The start statement initializes all queues in a schema to be empty.

A list (usually denoted by ℓ) is a structure as in LISP. The functions car, cdr, cons, and the predicate atom play the same role as in LISP (atom(x) is true if x is a data value, or Λ (nil), and false otherwise). The statements allowed are the following:

We use " lval " to represent Λ, a data variable, or a list variable,

(1) $\ell \leftarrow lval$

(2) $\ell \leftarrow cons(lval_1, lval_2)$

(3) if $\ell = \Lambda$ then goto L_1

(4a) $\underline{\text{if}}\ \text{atom}(\ell)\ \underline{\text{then}}\ \underline{\text{goto}}\ L_1$

$\underline{\text{else if}} \neg\ \text{atom}(\text{car}(\ell)) \vee \text{car}(\ell) = \Lambda\ \underline{\text{then}}\ \underline{\text{goto}}\ L_2$

$\underline{\text{else}}\ y_i \leftarrow \text{car}(\ell)$

(4b) $\underline{\text{if}}\ \text{atom}(\ell)\ \underline{\text{then}}\ \underline{\text{goto}}\ L_1$

$\underline{\text{else if}} \neg\ \text{atom}(\text{cdr}(\ell)) \vee \text{cdr}(\ell) = \Lambda\ \underline{\text{then}}\ \underline{\text{goto}}\ L_2$

$\underline{\text{else}}\ y_i \leftarrow \text{cdr}(\ell)$

(5a) $\underline{\text{if}}\ \text{atom}(\ell_i)\ \underline{\text{then}}\ \underline{\text{goto}}\ L\ \underline{\text{else}}\ \ell_j \leftarrow \text{car}(\ell_i)$

(5b) $\underline{\text{if}}\ \text{atom}(\ell_i)\ \underline{\text{then}}\ \underline{\text{goto}}\ L\ \underline{\text{else}}\ \ell_j \leftarrow \text{cdr}(\ell_i)$

where ℓ, ℓ_i, ℓ_j represent list variables, and L, L_1, L_2 represent labels. The start statement of a schema initializes all list variables to Λ (nil) . The class of schemas with lists is $\mathcal{C}(\text{list})$.

An array (A) is a one-dimensional, semi-infinite sequence of "locations" that can take on data and boolean values, and can be accessed by subscripting the array with a counter. The statements allowed are:

(1) $\langle y, z \rangle \leftarrow A[c]$,

and

(2) $A[c] \leftarrow \langle y, z \rangle$,

where A is an array, c is a counter, y is any data variable, and z is any boolean variable. In addition, the start statement is changed to initialize all arrays. It has the form

$$\text{START}\ \langle y_1, \ldots, y_n, z_1, \ldots, z_m \rangle \leftarrow \langle \tau_1(), \ldots, \tau_n(), \alpha_1(), \ldots, \alpha_m() \rangle$$

$$\langle A_1, \ldots, A_k \rangle \leftarrow \langle \tau'_1(), \alpha'_1(), \ldots, \tau'_k(), \alpha'_k() \rangle$$

where $A_1, \ldots, A_k$ are all the arrays used in the schema. The start

statement initializes all data locations of an array A_j to $\tau_j^!()$, and all boolean locations to $\alpha_j^!()$. The class of schemas with arrays is denoted $\mathcal{C}(A)$, and arrays with equality by $\mathcal{C}(A,=)$, etc. Note: the use of an array implies the use of counters, i.e., schemas in $\mathcal{C}(A)$ do have an arbitrary number of counters.

2.1.3 Recursive Schemas

A recursive schema is a set of mutually recursive function definitions (of defined functions $F_0, F_1, \ldots$). The functions are passed a vector of data and boolean arguments (the simple case -- "call by value" -- is assumed even though it does not always lead to the least fixed point: see Morris [1968], and also Cadiou [1972]), and they are allowed to return a vector of values.

Given a vector $\langle \bar{y}, \bar{z} \rangle$ of data values $\bar{y} = y_1, y_2, \ldots, y_n$, and boolean values $\bar{z} = z_1, z_2, \ldots, z_m$, we define the notation for picking off the i-th data or boolean values as follows:

$$Y_i(\bar{y},\bar{z}) = y_i \quad \text{and} \quad Z_i(\bar{y},\bar{z}) = z_i$$

provided i does not exceed the maximum index (in either case). If a vector has n data values and m boolean values, we say its <u>type</u> is (n,m). A vector of type $(1,0)$ is a data element, and a vector of type $(0,1)$ is a boolean element.

We can now define a recursive schema. It is a set of definitions of the form:

$$F_0 \Leftarrow \tau_0(\bar{F});$$

$$F_1(\bar{y},\bar{z}) \Leftarrow \underline{\text{if}}\ \alpha_1(\bar{y},\bar{z},\bar{F})\ \underline{\text{then}}\ \bar{\tau}_1(\bar{y},\bar{z},\bar{F})\ \underline{\text{else}}\ \bar{\tau}'_1(\bar{y},\bar{z},\bar{F});$$

$$F_2(\bar{y},\bar{z}) \Leftarrow \underline{\text{if}}\ \alpha_2(\bar{y},\bar{z},\bar{F})\ \underline{\text{then}}\ \bar{\tau}_2(\bar{y},\bar{z},\bar{F})\ \text{else}\ \bar{\tau}'_2(\bar{y},\bar{z},\bar{F});$$

$$\vdots$$

$$F_k(\bar{y},\bar{z}) \Leftarrow \underline{\text{if}}\ \alpha_k(\bar{y},\bar{z},\bar{F})\ \underline{\text{then}}\ \bar{\tau}_k(\bar{y},\bar{z},\bar{F})\ \underline{\text{else}}\ \bar{\tau}'_k(\bar{y},\bar{z},\bar{F});$$

where $\bar{F} = \langle F_1,F_2,\ldots,F_k\rangle$ and $\bar{y},\bar{z}$ represent arbitrary vector arguments in each case, τ_0 is of type data, and α_i is of type boolean. Terms can be constructed using the arguments $\bar{y},\bar{z}$ of the defined function, and applying the base functions, defined functions, and the notation Y_i,Z_i for extracting an element from a vector. It is implicitly assumed that there is no type mismatch.

The computation rule for terms in the schema is leftmost innermost, with the exception that if exactly the same function call appears more than once in a function definition it will not be computed more than once -- rather, the values returned by the first call are substituted in the others (in fact we could have prevented multiple identical terms from appearing by a more complicated notation). This is one of the reasons for allowing functions to return vectors, i.e., it results in relatively efficient computations. For example, consider the schema S below (unnecessary parentheses are omitted):

S: $F_0 \Leftarrow h(Y_1F_1(a,a),Y_2F_1(a,a))$;

$F_1(y_1,y_2) \Leftarrow \underline{\text{if}}\ p(y_1)\ \underline{\text{then}}\ \langle y_1,y_2\rangle$

$\underline{\text{else}}\ \langle fY_1F_1(fy_1,gy_2),Y_2F_1(fy_1,gy_2)\rangle$.

Not calling F_1 both times in $\langle fY_1F_1(fy_1,gy_2),Y_2F_1(fy_1,gy_2)\rangle$ results in an exponential saving in the length of the computation.

The class of recursive schemas will be denoted $\mathcal{C}(R)$. The number of "variables" in a recursive schema is the maximum number of data elements either passed as arguments to, or returned from, a defined function. The class of recursive schemas in which no defined function is passed more than n data variables, and no function returns more than n data values is denoted $\mathcal{C}(R,n\ var)$; similarly, the class of recursive schemas which allow equality tests is denoted $\mathcal{C}(R,=)$, etc.

In the rest of Section 2.1 whenever we refer to an arbitrary uninterpreted schema we mean a schema from $\mathcal{C}(pds,q,list,A,=) \cup \mathcal{C}(R,=)$. We can get an interpreted schema by restricting the interpretations allowed. One way of doing this is by specifying that every interpretation for a schema satisfy some formula in predicate calculus; but mostly the schemas we consider will be uninterpreted.

2.1.4 Halting, Divergence, and Freedom

Definition. A schema is said to halt if it halts on every interpretation.

Definition. A schema is said to diverge if it diverges on every interpretation, that is, it does not halt on any interpretation.

Definition. Let $s_0, s_1, s_2, \ldots$ be the statements of a flowchart, or an augmented flowchart schema S. Then, a path in S is defined to be a finite or infinite sequence

$$\langle t_0, t_1, t_2, \ldots \rangle$$

where for each i, t_i is s_j for some j, if s_j is a start, halt, loop, or an assignment statement, or t_i is $\langle s_j, \text{true} \rangle$, or $\langle s_j, \text{false} \rangle$ if s_j is a test statement, and the sequence must have the property that

(i) t_0 is the start statement, and no other t_i is the start statement,

(ii) only the last element in the sequence (if any) can be a halt or a loop statement,

(iii) if t_i is the start statement, or assignment statement, then t_{i+1} corresponds to the statement following t_i in the schema,

(iv) if t_i is $\langle s_j, \text{true} \rangle$ then t_{i+1} corresponds to the statement following the test s_j if it takes the true exit; and similarly for $\langle s_j, \text{false} \rangle$.

Definition. We can similarly define the notion of a path in a recursive schema. Let S be a recursive schema, and $F_0, F_1, F_2, \ldots$ be its defined functions, and $s_1, s_2, \ldots$ be the corresponding tests in the if-then-else definitions. Then a path in S is a finite or infinite sequence

$$\langle t_0, t_1, t_2, \ldots \rangle$$

where for each i, t_i is either $\langle \text{enter } F_j \rangle$, $\langle \text{exit } F_j \rangle$, $\langle s_j, \text{true} \rangle$, or $\langle s_j, \text{false} \rangle$. The first element, t_0, is $\langle \text{enter } F_0 \rangle$, and only the last element, if any, can be $\langle \text{exit } F_0 \rangle$. The significance of the t_i's

is obvious, and we say that a path must have the property that the sequence of t_i's must obey the computation rule for recursive schemas, (that is, leftmost innermost, with substitutivity for identical terms in the same function definition).

Definition. Given a schema S and an interpretation I for S, the path of the computation of S on I is denoted by Path(S,I).

Definition. A schema is said to be free if every path in the schema can be taken by its computation on some interpretation.

As example, the schema S_a is not free because the path $\langle s_0, \langle s_1, \text{false}\rangle, \langle s_2, \text{true}\rangle\rangle$ cannot be taken for any interpretation. In fact, even the schema S_b is not free because no interpretation can take the false-exit from statement L_3 (even though the true-exit and the false-exit both lead to the same statement). The schema S_c is free, as is the recursive schema S_d. However, the recursive schema S_e is not free because the test $F_2(y)$ can only take the true exit.

```
S_a:    START y ← a;              comment: call this statement s_0;
   L_1: if p(y) then goto L_2;    comment: call this s_1;
        if p(a) then goto L_1;    comment: call this s_2;
   L_2: HALT(y) ;                 comment: call this s_3;

S_b:    START y ← a;              comment: call this statement s_0;
        if p(y) then goto L;      comment: call this s_1;
        if p(a) then goto L;      comment: call this s_2;
     L: HALT(y) .                 comment: call this s_3;
```

S_c: START y ← a;

while p(y) do y ← f(y);

HALT(y) .

S_d: $F_0 \Leftarrow F_1(a)$;

$F_1(y) \Leftarrow$ if $F_2(y)$ then f(y) else g(y);

$F_2(y) \Leftarrow$ if p(y) then $F_2(f(y))$ else $F_2(g(y))$.

S_e: $F_0 \Leftarrow F_1(a)$;

$F_1(y) \Leftarrow$ if $F_2(y)$ then f(y) else g(y);

$F_2(y) \Leftarrow$ if p(y) then true else $F_2(g(y))$.

Freedom, as defined, is not a very useful concept for augmented schemas because some of the functions and tests are totally interpreted. Thus, if a counter schema tests " c = 0 ", then all paths in the schema cannot be taken because the outcome of this test is fixed once we fix a path leading to this test. The same is true, for example, for a stack (a schema attempting to pop a stack must test if it is empty), a queue, or a list.

2.1.5 Equivalence

Given a schema S and an interpretation I for S we use the nctation Val(S,I) to denote the output (of the computation) of S on I -- if S does not halt, then Val(S,I) is undefined.

Definition. Given two (uninterpreted) schemas S_1 and S_2, we say that S_2 includes S_1 ($S_1 \leq S_2$) if for every interpretation I for

S_1 and S_2 (that is, I specifies all base functions and predicates used in both S_1 and S_2), if $Val(S_2,I)$ is defined, then so is $Val(S_1,I)$ and $Val(S_1,I) = Val(S_2,I)$.

Definition. Two schemas S_1 and S_2 are said to be equivalent ($S_1 \equiv S_2$) if $S_1 \leq S_2$ and $S_2 \leq S_1$; that is, for all interpretations I for S_1 and S_2, if one schema halts, then so does the other with the same output.

The notion of equivalence ($\equiv$) is sometimes also called output equivalence, or strong equivalence.

It is immediate that the relation $\equiv$ is reflexive and symmetric. It is also transitive, but this proof requires a little care. The only problem is that given $S_1 \equiv S_2$ and $S_2 \equiv S_3$, to show that $S_1 \equiv S_3$ we have to show that if I is any interpretation for S_1 and S_3 then $Val(S_1,I) = Val(S_3,I)$. But I may not be an interpretation for S_1 and S_2 (or for S_2 and S_3, for that matter) because S_2 may contain some superfluous functions or predicates. To overcome this problem, we note that if I' is any interpretation for S_1, S_2 and S_3, then $Val(S_1,I') = Val(S_2,I') = Val(S_3,I')$. And from this, the desired result follows, for if I is any interpretation for S_1 and S_3, we can extend it to I' by adding the new functions and predicates of S_2 (arbitrarily) and then $Val(S_1,I) = Val(S_1,I') = Val(S_3,I') = Val(S_3,I)$.

An alternative definition of equivalence (and a corresponding one applies to inclusion) is that $S_1 \equiv S_2$ if for every interpretation I_1 for S_1 there is an isomorphic interpretation I_2 for S_2 (let θ denote the isomorphism $\theta: I_1 \leftrightarrow I_2$, i.e., θ is a one-one mapping from the

domain of I_1 onto the domain of I_2 that preserves functions and predicates) such that if $Val(S_1,I_1)$ or $Val(S_2,I_2)$ is defined, then both are defined, and $Val(S_2,I_2) = \theta(Val(S_1,I_1))$.

The two definitions are the same owing to a basic notion regarding schemas -- that the behavior of a schema over two isomorphic interpretations is the same, i.e., the paths of computation are the same, and the values of all variables correspond under the isomorphism at each step.

A fundamental notion of equivalence is that if we want to find a schema equivalent to some given schema S , then the schema to be found need not have any function or predicate symbol other than those in S . This result is implicitly used all the time in the theory of schemas, apparently without ever having been clearly formalized.

Theorem 2.1 (Redundant predicates and functions)

Given uninterpreted schemas S and S_1 such that $S \equiv S_1$, we can find a schema S_2 equivalent to S such that

(i) S_2 has no function or predicate symbol not in both S and S_1 ,

(ii) S_2 has exactly the same features (that is, equality, recursion, number of variables, counters, stacks, queues, lists and arrays) as those of S_1 .

This theorem may also be called the "interpolation lemma for program schemas".

For a proof, see Section 2.1.9. In this connection it may be remarked that if we are given any schema S and a flowchart schema S_1

$(S_1 \in \mathcal{C}(=))$ equivalent to S, then there exists a schema S_2 also equivalent to S having properties (i), (ii) above, and also preserving the freedom of S_1, i.e., (iii) S_2 is free if and only if S_1 is free. This, in itself, is not astonishing. But it should be noted that we said "there exists a schema S_2", not that we can find it (as in the theorem). It may come as a surprise that if we wish to preserve freedom, then S_2 cannot be effectively found in general! This is demonstrated along with the proof of this remark in Section 2.1.9.

Definition. We say a class of schemas $\mathcal{C}_2$ is more powerful than another class $\mathcal{C}_1$ $(\mathcal{C}_1 \leq \mathcal{C}_2)$ if for every schema in $\mathcal{C}_1$ there is an equivalent schema in $\mathcal{C}_2$.

Note that the meaning of the symbol " $\leq$ " is quite different when applied to individual schemas, and when applied to classes of schemas.

Definition. A class $\mathcal{C}_2$ is strictly more powerful than $\mathcal{C}_1$ $(\mathcal{C}_1 < \mathcal{C}_2)$ if $\mathcal{C}_1 \leq \mathcal{C}_2$, but not $\mathcal{C}_2 \leq \mathcal{C}_1$.

Definition. Two classes $\mathcal{C}_1$ and $\mathcal{C}_2$ are equally powerful, or equipollent, $(\mathcal{C}_1 \equiv \mathcal{C}_2)$ if $\mathcal{C}_1 \leq \mathcal{C}_2$, and $\mathcal{C}_2 \leq \mathcal{C}_1$.

2.1.6 Isomorphism

Intuitively, saying that two schemas are isomorphic means that they perform their computations in the same fashion. This differs from equivalence which says that two schemas always produce the same output

even though they might perform their computations by quite different algorithms; for example, one schema might be more efficient than another as far as the number of operations is concerned.

Several notions of isomorphism can be defined. We consider some of these possibilities informally before presenting our definition. The strongest notion, of course, is the identity relation between schemas. A weaker definition (call it N_1) is that two schemas are isomorphic if both compute exactly the same statements (i.e., statements that look the same) in the same order, for each interpretation. Under this notion, if the roles of two variables are interchanged the schemas are not N_1-isomorphic, as in the case of the two schemas S_f, S_g below:

S_f: START $\langle y_1, y_2 \rangle \leftarrow \langle a, a \rangle$;
$y_1 \leftarrow f(y_2)$;
HALT(y_1)

S_g: START $\langle y_1, y_2 \rangle \leftarrow \langle a, a \rangle$;
$y_2 \leftarrow f(y_1)$;
HALT(y_2) .

A weaker notion (call it N_2) is that two schemas are isomorphic if the same terms are computed (in the same order). Thus the schemas S_f, S_g above are N_2-isomorphic because both compute the term $f(a)$ only. But the schemas S_h, S_i below are not N_2-isomorphic:

S_h: START $y \leftarrow a$;
$y \leftarrow fg(y)$
HALT(y)

S_i: START $y \leftarrow a$;
$y \leftarrow g(y)$;
$y \leftarrow f(y)$;
HALT(y)

because S_h computes $fg(a)$ and S_i computes first $g(a)$, and then $fg(a)$. A weaker notion (N_3) breaks down the computation of terms into

its constituent parts making S_h , S_i N_3-isomorphic, but not S_j , S_k below:

S_j: START y ← a;
if y = b then y ← f(a);
HALT(y)

S_k: START y ← a;
if y = b then y ← f(b);
HALT(y) .

The definition given below is a still weaker (and to us, a more reasonable) definition that operates on elements of the interpretation rather than on terms. It should be stated, however, that the decidability and undecidability results of the next chapter remain unchanged if any of the notions N_1 , N_2 , or N_3 is substituted instead.

Definition. Given a schema S and an interpretation I for S . Let Seq(S,I) denote the (possibly infinite) sequence of vectors of the form

$\langle f, e_1, \ldots, e_k \rangle$ -- where f is a k-ary function symbol,

$\langle p, e_1, \ldots, e_k \rangle$ -- where p is a k-ary predicate symbol,

$\langle =, e_1, e_2 \rangle$,

$\langle$HALT$, e_1 \rangle$, or

$\langle$LOOP$\rangle$

where the e_i's are elements in the domain of I -- that are evaluated during the computation of S on I .

For example, if for the schema S_j above, I is over the domain $\{1,0,2,\ldots\}$, a = b = 0 , and f is the "add-one" function, then $Seq(S_j,I) = \langle\langle =,0,0\rangle,\langle f,0\rangle,\langle$HALT$,1\rangle\rangle = Seq(S_k,I)$.

Definition. Two schemas S_1 and S_2 are isomorphic (denoted $S_1 \sim S_2$ or $S_1 \underset{\text{isom}}{\equiv} S_2$) if for every interpretation I ,

$Seq(S_1,I) = Seq(S_2,I)$.

It is obvious from the definition that if two schemas are isomorphic then they are equivalent. The converse, of course, is not true.

2.1.7 Herbrand Schemas

Definition. Given a set of function symbols (containing at least one zero-ary symbol) and predicate symbols, a Herbrand interpretation on the set is defined as follows: the domain is the set of (fully parenthesized) constant terms using the function symbols; the functions are defined in the usual way for terms, and predicates are arbitrary.

An example may help clarify the definition. Given the set of symbols $\{a,f,g,p\}$ where a is a zero-ary function symbol, f and g are unary function symbols, and p is a unary predicate symbol, a Herbrand interpretation for this set has the infinite domain

$$\{\text{"a"},\text{"f(a)"},\text{"g(a)"},\text{"f(f(a))"},\text{"f(g(a))"},\ldots\}$$

where, for example, by "f(a)" we mean the term f(a) itself, consisting of a string of four symbols -- "f" , "(" , "a" , and ")" . In the interpretation, the value of the function f applied, for example, to the element "f(a)" is the element "f(f(a))" , and similarly for g . The value of p applied to any element in the domain can be arbitrarily true or false.

Definition. Given an interpretation I over a set of function and predicate symbols, the Herbrand interpretation I^H corresponding to I is a Herbrand interpretation whose predicates are defined as follows: if p is a k-ary predicate symbol, and $\tau_1, \tau_2, \ldots, \tau_k$ are (fully parenthesized) constant terms, then $p(\tau_1, \tau_2, \ldots, \tau_k)$ is true in I^H if and only if it is true in I.

As an example, consider the set of symbols $\{a,f,g,p\}$, and let I be an interpretation with domain $\{0,1\}$ such that $a = 0$, $f(x) = x$, $g(x) = 1-x$, and $p(x)$ is true for $x = 0$ and false for $x = 1$. Then I^H is over the infinite domain mentioned above, and $p("a")$, $p("f(a)")$, $p("g(g(a))")$ etc., are true, and $p("g(a)")$, $p("f(g(a))")$ etc., are false. In general, $p(y)$ is true if y has an even number of g's.

Definition. An uninterpreted schema S is said to be a Herbrand schema if for every interpretation I for S, $\text{Path}(S,I) = \text{Path}(S,I^H)$.

In Chapter 4 this definition is extended to interpreted schemas as well.

Definition. An inherently non-Herbrand schema is a non-Herbrand schema for which there is no equivalent Herbrand schema.

Examples are given below (schemas $S_1 - S_0$).

The following simple but very useful theorem indicates why the notion of Herbrand schemas is useful. We say that a schema S is free

on a set of interpretations $\mathcal{I}$ if for every path in S there is some interpretation in $\mathcal{I}$ on which the computation follows that path; a schema S halts (or diverges) on $\mathcal{I}$ if it halts (diverges) for every interpretation in $\mathcal{I}$; we say that $S_1 \leq S_2$ on $\mathcal{I}$ if for every $I \in \mathcal{I}$, if $Val(S_1,I)$ is defined then $Val(S_1,I) = Val(S_2,I)$; and similar definitions apply for equivalence and isomorphism. We use $\mathcal{H}$ to denote the class of Herbrand interpretations.

Theorem 2.2 (Fundamental theorem of Herbrand schemas)

If S_1 and S_2 are uninterpreted Herbrand schemas then

(a) S_1 halts if and only if S_1 halts on $\mathcal{H}$,

(b) S_1 diverges if and only if S_1 diverges on $\mathcal{H}$,

(c) $S_1 \equiv S_2$ if and only if $S_1 \equiv S_2$ on $\mathcal{H}$,

(d) $S_1 \leq S_2$ if and only if $S_1 \leq S_2$ on $\mathcal{H}$,

(e) $S_1 \sim S_2$ if and only if $S_1 \sim S_2$ on $\mathcal{H}$,

(f) S_1 is free if and only if S_1 is free on $\mathcal{H}$.

Parts (a), (b), and (f) are immediate from the definition of Herbrand schemas; and part (c) follows from (d). For proofs of (d) and (e) see Section 2.1.9.

We would now like to know what kinds of schemas are Herbrand schemas. The next theorem implies that it is the tests of equality that tend to make schemas non-Herbrand.

Theorem 2.3 (Schemas without equality are Herbrand)

If S is an uninterpreted schema without any equality test then S is a Herbrand schema.

Thus, the schemas in $\mathcal{C}()$, $\mathcal{C}$(n var) , $\mathcal{C}$(pds,q,list,A) , $\mathcal{C}$(R) , etc., are all Herbrand schemas. In general, however, it is not partially solvable if a given schema is a Herbrand schema. This follows directly from the fact (see, for example, Luckham, Park and Paterson [1970]) that the divergence problem for $\mathcal{C}$(2 var) is not partially solvable. This is so because if we are given a schema $S \in \mathcal{C}$(2 var) and we replace all halt statements in S by

if a = b then HALT(y) else HALT(y)

(where a,b are zero-ary functions not present in S) to get a schema in $\mathcal{C}$(2 var, =) , call it S' , then S' is a Herbrand schema if and only if S diverges.

Examples. Consider the schema S_ℓ below:

S_ℓ: START $y \leftarrow a_1$;

if $a_1 = a_2$ then HALT(y) else LOOP .

This is a non-Herbrand schema because for every Herbrand interpretation $a_1 \neq a_2$, though a_1 can equal a_2 for some non-Herbrand interpretations. In fact, S_ℓ is an inherently non-Herbrand schema, because if there is a Herbrand schema, say S'_ℓ , equivalent to S_ℓ , then S'_ℓ loops for all Herbrand interpretations. But consider an interpretation I for which S_ℓ halts, then S'_ℓ too must halt for I , and hence must also halt for the Herbrand interpretation corresponding to I (since S'_ℓ is a Herbrand schema by hypothesis) -- a contradiction.

However, the use of equality tests does not necessarily make a schema inherently non-Herbrand, or even non-Herbrand. S_m is a Herbrand schema that uses equality tests. It is equivalent to a (Herbrand) schema without any equality tests (S_n) and also to a non-Herbrand schema (S_o) with equality tests!

```
S_m:   START ⟨y_1,y_2⟩ ← ⟨a,a⟩;
    L:if p(y_1) then
          if p(y_2) then
              begin y_1 ← f(y_1);
                    y_2 ← f(y_2);
                    goto L;
              end
          else if y_1 = a then HALT(y) else LOOP
      else if y_1 = y_2 then HALT(y) else LOOP .

S_n:   START y ← a;
    L:if p(y) then
          begin y ← f(y);
                goto L
          end
      else HALT(y) .

S_o:   START y ← a;
    L:if p(y) then
          if y = f(y) then LOOP
          else begin y ← f(y);
                     goto L;
               end
      else HALT(y) .
```

2.1.8 Value Languages

Given a (fully parenthesized) term τ, let $[\tau]$ denote the string τ with all parentheses and all zero-ary function symbols removed. For example, $[f(g(f(a)))] = fgf$.

Definition. Given a schema S, let $\mathcal{H}$ denote the set of Herbrand interpretations for S, then the value language L(S) of the schema S is defined by

$$L(S) = \{[\tau] \mid \exists H \in \mathcal{H}, \mathrm{Val}(S,H) = \tau\} .$$

For example, the value language of the recursive schema S_p is $L(S_p) = \{xx^R \mid x \in \{f,g\}^*\}$ where x^R means the reverse of the string x.

S_p: $F_0 \Leftarrow F_1(a)$;
$F_1(y) \Leftarrow$ if $p(y)$ then y else $F_2(y)$;
$F_2(y) \Leftarrow$ if $q(y)$ then $fF_1f(y)$ else $gF_1g(y)$;

Theorem 2.4 (Value languages are r.e.)

The value language of any schema S (that admits all the Herbrand interpretations $\mathcal{H}_S$) is recursively enumerable.

The proof is quite simple, and is given in Section 2.1.9.

Value languages have been studied mostly for monadic schemas. They can be used to prove theorems regarding the power of classes of schemas. The following lemma is a slight generalization of one given by Garland and Luckham [1971].

Theorem 2.5 (Basic theorem of value lanuages)

For uninterpreted schemas S_1 , S_2 , if $S_1 \leq S_2$ then $L(S_1) \subset L(S_2)$.

The proof is trivial, for if $L(S_1) \not\subset L(S_2)$ then there is a string $x \in L(S_1)$ such that $x \notin L(S_2)$. Now, consider any Herbrand interpretation H for S_1 for which $[Val(S_1,H)] = x$, then $[Val(S_2,H)] \neq x$ because $x \notin L(S_2)$, and hence $S_1 \not\leq S_2$.

Note that this theorem holds whether or not the schemas S_1 , S_2 are Herbrand schemas.

Corollary 2.6. For schemas S_1 , S_2 , if $S_1 \equiv S_2$ then $L(S_1) = L(S_2)$.

This is usually used to prove the negative result: given two classes $\mathcal{C}_1$ and $\mathcal{C}_2$ of uninterpreted schemas such that for some $S_1 \in \mathcal{C}_1$ there is no $S_2 \in \mathcal{C}_2$ for which $L(S_1) = L(S_2)$ then we can conclude that $\mathcal{C}_1 \not\leq \mathcal{C}_2$.

2.1.9 Discussion and Proofs

2.1.9.1 On the Treatment of Equality

In our treatment, equality is viewed as a basic construct in schemas, on par with others like assignments, goto statements (in flowchart notation, the arrows leading from one statement to another) or the use of more than one variable in schemas.

Alternatives have been suggested, but our approach seems to be the most natural. One alternative is to treat equality as just another (diadic) base predicate, call it $p_=$. Then, a test like $\tau_1 = \tau_2$ is viewed as just a notation for the strict form $p_=(\tau_1,\tau_2)$. However, the

schema is no longer uninterpreted, but every interpretation must satisfy the formula $\forall x \forall y\ p_{=}(x,y) \equiv (x = y)$. In other words, $p_{=}$ is treated as pseudo-equality. The problem is that the equivalence of partially interpreted schemas has to be defined (it is not desirable to define it for the special cases where zero or one of the predicates is pseudo-equality). The definition of Section 2.1.5 (i.e., S_1 and S_2 are equivalent if $\forall I$ if S_1 admits I , and S_2 admits I then $Val(S_1,I) = Val(S_2,I)$) is inadequate because it is not transitive in general. Equivalence is defined in Chapter 4 for partially interpreted schemas (it is based on the alternative definition given in Section 2.1.5). If this definition is used, we would find that the trivial schemas S_1 and S_2 below are not equivalent using the $p_{=}$ formalism, while clearly we would like to say that they are indeed equivalent. In fact we would find that the uninterpreted schema S_2 is a "generalization" (see Section 4.3) of S_1 because more interpretations are allowed for S_2 than for S_1 . It may be noted that S_1 and S_2 are equivalent in our formalism.

S_1: START $y \leftarrow a_1$;
<u>if</u> $a_1 = a_2$ <u>then</u> HALT(a_1) <u>else</u> HALT(a_2) .

S_2: START $y \leftarrow a_1$;
HALT(a_2) .

Another approach that has been suggested is to treat equality as just a (diadic) base predicate, say $q_{=}$. The schema is to be partially interpreted, with $q_{=}$ being an equivalence relation also satisfying substitutivity; i.e., if $f_1,f_2,\ldots,f_r$ and $p_1,p_2,\ldots,p_s$ are the other base functions and predicates in a schema with ranks $i_1,\ldots,i_r$ and $j_1,\ldots,j_s$ respectively (let k be the maximum of these), then

every interpretation for the schema is to satisfy the formula φ, where

$$\begin{aligned}
\varphi \text{ is } \forall x_1 \forall x_2 \forall x_3 \quad & q_=(x_1,x_1) \\
& \wedge q_=(x_1,x_2) \rightarrow q_=(x_2,x_1) \\
& \wedge q_=(x_1,x_2) \wedge q_=(x_2,x_3) \rightarrow q_=(x_1,x_3) \\
\wedge \forall x_1 \ldots \forall x_k \quad & (q_=(x_1,y_1) \wedge \ldots \wedge q_=(x_k,y_k)) \rightarrow \\
& \qquad q_=(f_1(x_1,\ldots,x_{i_1}),f_1(y_1,\ldots,y_{i_1})) \\
& \wedge \ldots \\
& \wedge q_=(f_r(x_1,\ldots,x_{i_r}),f(y_1,\ldots,y_{i_r})) \\
& \wedge p_1(x_1,\ldots,x_{j_1}) \equiv p_1(y_1,\ldots,y_{j_1}) \\
& \wedge \ldots \\
& \wedge p_s(x_1,\ldots,x_{j_1}) \equiv p_s(y_1,\ldots,y_{j_s}) \quad .
\end{aligned}$$

This approach "works" for the introduction of equality in, say, first order predicate calculus where the property of interest is the validity of formulas -- a formula ψ with equality is valid (satisfiable) if and only if $\psi' \wedge \varphi$ is valid (satisfiable) where ψ' is obtained from ψ by substituting $q_=$ for equality. Unfortunately, this approach does not seem to be viable for schemas, where the equivalence of schemas should be preserved on replacement of equality by $q_=$. Observe that the schemas S_1 and S_2 are not equivalent if $a_1 = a_2$ is replaced by $q_=(a_1,a_2)$ in S_1, because it is possible for a_1 and a_2 to be distinct elements even if $q_=(a_1,a_2)$ is true, i.e., the outputs of S_1 and S_2 are not the same. Of course, the outputs are equivalent under the relation $q_=$ for every interpretation, but as mentioned, equivalence of schemas should be defined for some general class and not for a special case where there is one equivalence relation.

Why all this discussion on equality? It goes back to the basic question "what is a program schema". The intuitive notion is that of a machine that computes on uninterpreted (or partially interpreted) domains, as against "real" computations on interpreted domains. One aim of the study is to present stable (or "maximal") classes of machines similar to the Turing machines for real computations. What properties should schemas possess? As with real computations, the requirements of finiteness, nonrandomness, and discreteness seem reasonable -- see e.g. Rogers [1967]. In addition we may require the following:

(1) first order functions and predicates;

(2) total functions and predicates;

(3) the computation of a schema should be fully characterized by an interpretation (and the inputs, if any);

(4) computations on isomorphic interpretations must be the "same" for any one schema;

(5) in any one step a schema should be able to "look at" at most a finite number of elements of the domain of the interpretation.

Of course, one may relax any of these conditions to study what classes of machines are obtained. In Chapter 4 we introduce a class of schemas having all the above properties. In addition, a slightly stronger version of (3) above is used: the computation of a schema is fully characterized by the values of the functions and predicates applied to the reachable elements in the domain -- the set of reachable elements is the smallest set (containing the inputs, if any, and) closed under function applications. In this class of schemas we obtain a maximal subclass

for the uninterpreted schemas, and a maximal subclass for the uninterpreted Herbrand schemas (i.e., schemas whose computation is the same for any interpretation and its corresponding free interpretation), and as may be expected, the use or the non-use of equality plays a crucial role in distinguishing the subclasses.

2.9.9.2 Proof of Theorem 2.1 (Redundant functions and predicates)

Proof of the Theorem

Given uninterpreted schemas S, S_1 such that $S \equiv S_1$, then there is a schema S_2 equivalent to S, having no function or predicate symbol other than those in both S and S_1, and having exactly the same features as S_1.

Proof. Firstly, if there is no zero-ary function symbol common to both S and S_1 then both must diverge for all interpretations because if not, consider the interpretations for S and S_1 -- as the sets of terms generated by S and S_1 are mutually disjoint, if S halts on any interpretation then it halts on one in which the reachable elements of S and of S_1 are disjoint, and for this interpretation the output of S_1 can never equal that for S. So in this case the construction of S_2 is trivial.

Now, if S and S_1 have a common zero-ary function, say a, then we obtain S_2 from S_1 as follows: if f is any (k-ary) function in S_1 and not in S, then replace any term of the form

$$f(\tau_1,\ldots,\tau_k) \text{ by } a,$$

and if p is any (k-ary) predicate of S_1 not in S, then replace any atomic formula

$$p(\tau_1,\ldots,\tau_k) \text{ by true}.$$

Now, to prove that $S \equiv S_2$, let I be any interpretation for S and S_2. We change I to I' by first deleting all functions and predicates of $\Sigma(S_1) - \Sigma(S_2)$ from I (if any), and then adding the functions and predicates of $\Sigma(S_1) - \Sigma(S_2)$ as follows: the value of each new function f applied to any set of elements in the domain is "a", and all new predicates are "true" for all arguments. Clearly, $Val(S,I') = Val(S,I)$ and $Val(S_2,I') = Val(S_2,I)$ because the functions and predicates of $\Sigma(S_1) - \Sigma(S_2)$ do not appear in S or S_2. Also, on I', the computations of S_1 and S_2 are identical, and hence $Val(S,I') = Val(S_1,I') = Val(S_2,I')$. This gives the desired result, i.e., $Val(S,I) = Val(S_2,I)$.

□

Redundant functions and predicates with preservation of freedom

Given a schema S and a flowchart schema S_1 $(S_1 \in \mathcal{C}(=))$ equivalent to S, then there exists another flowchart schema S_2 also equivalent to S having the same features as S_1 and no base functions or predicates other than those in both S and S_1, such that S_2 is free if and only if S_1 is free. But S_2 cannot be effectively found, in general.

Proof. $S \equiv S_1$, $S_1 \in \mathcal{C}(=)$. We first construct a flowchart schema S_1' equivalent to S and having no base functions and predicates other than those in S, such that S_1' is free if S_1 is free (but it may also be free if S_1 is not).

The idea behind the construction is similar to that in the proof of the theorem. The application of any new predicate p (p is in S_1, but not in S) yields "true", and the value of any new function f is a special element we call "bad". The schema S_1' simulates the computation of S_1, keeping track of all "bad" variables. S_1' can be described as follows. It has $2^{n}3^{m}$ "copies" of S_1 -- where n is the number of data variables, and m is the number of boolean variables. Each data variable can be good, or bad, each boolean variable can be good, bad-true, or bad-false. If in S_1 there is an assignment

$$y_i \leftarrow \tau \quad \text{or} \quad z_i \leftarrow \alpha$$

where τ (or α) contains a bad value (for some copy in S_1') or a new predicate, then this assignment is not made (in that copy), but the variable becomes bad, i.e., S_1' transfers to the appropriate next statement. Further, if z_i becomes bad, the value it takes is governed

by the rule that any predicate on the value "bad" is true, and "bad = bad" yields true, but "bad = good" yields false (where "good" stands for some term that is not bad). The same applies to any bad test -- the test is not actually made, but the appropriate exit is assumed.

Now it is easy to see that $S_1' \equiv S_1$. The proof is very similar to the proof for the theorem (above).

Further, S_1' is free if S_1 is free. Suppose S_1' is not free. Then there is some path from the start statement to a test such that the outcome of the test is predetermined by the path. But as S_1' makes tests only on (constant) terms that can only be obtained by applications of functions of S, we see that in the corresponding path in S_1, any computation following this path must take the same exit. This is so because (a) any interpretation of the form having the "bad" element appended, must take the same exit, and (b) for any interpretation I, we can obtain the corresponding interpretation B with a "bad" element, such that if I follows the path, then its exit is the same as that of B.

Now, if the given schema S_1 is free, then S_1' is the required schema S_2, otherwise to obtain S_2 we can simply append to the beginning of S_1' some trivial tests to force it to be non-free. □

Unsolvability of the translation

Our translation was not effective because in the last step the decision as to whether S_1 is free or not was not effective.

We will prove that the translation to S_2 is not solvable in general in a very informal way. We use Paterson's proof [1967] of the unsolvability of freedom and convert it to the unsolvability of freedom for schemas in $\mathcal{C}$(1 var, =) by using the method of simulating two variables with only one presented in the proof of Theorem 3.3. The resulting class (call it $\mathcal{C}'$) has schemas with no predicate, one zero-ary function a, and unary functions, one of which is called f. There is a single variable y which, at intervals, takes values

$$a, f(a), ff(a), fff(a), \ldots .$$

We will change this class $\mathcal{C}'$ somewhat to $\mathcal{C}_1$ by adding a unary predicate p, and whenever in a schema $S' \in \mathcal{C}'$ the variable y has value $f^i(a)$ in the above sequence, the new schema S_1 makes a test $p(y)$. If $p(y)$ is false, the schema S_1 halts, otherwise it continues like S'. In addition, any halt or loop statement in S' is replaced by a cycle that tests

$$p(f^i(a)), p(f^{i+1}(a)), p(f^{i+2}(a)), \ldots$$

such that S_1 halts if any of them is false. Now, S_1 is free if and only if S' is free, and hence the freedom problem for this new class is unsolvable. But, each schema S_1 in this class $\mathcal{C}_1$ is equivalent to the schema S:

S = START y ← a;

<u>while</u> p(y) <u>do</u> y ← f(y);

HALT(y) .

Hence, if our desired schema S_2 exists, it must have one variable y, functions a and f, and predicate p. But the freedom problem for

such a class of schemas can be shown to be solvable. We do not give a rigorous proof here, but only indicate it.

Given a flowchart schema S with only a zero-ary function a, one unary function f, and one unary predicate p, to show that the freedom problem for S is solvable we observe that without loss of generality we can assume that every circular path (cycle) in S must have at least one predicate or equality test.

Now, if any reset (i.e., $y \leftarrow f^i(a)$) appears in a cycle, then S must be nonfree for the same test would be made twice (with the same value for y) by going around the loop.

Secondly, if after the "true" exit from any equality test (i.e., $f^i a = f^j a$, $f^i a = f^j y$, or $f^i y = f^j y$) there is a cycle then the schema must be nonfree because either the false exit can never be taken, or else there are only a finite number n of distinct elements in $a, fa, f^2a, f^3a, \ldots$, and hence by going around the cycle n+1 times some test would be made twice.

Now, if the schema S is not obviously nonfree by the above criteria then we can determine whether or not it is free by constructing a finite state automaton that accepts all input tapes unless the schema is nonfree. We use the terminology in the proof of Theorem 3.1.

The input tape of the automaton represents a path through the schema. The first symbol specifies all resets the path goes through, and true exits from equality tests. Subsequent symbols update each of these subpaths starting from the resets and true exits. The automaton simulates the computation of all possible interpretations simultaneously along all these subpaths (except for any true exit from a $f^i y = f^j y$ test, which

is simulated when computation reaches that statement). Note that the number of equivalence classes of all interpretations remains bounded. The input tape is accepted unless it represents a valid path which cannot be traced by an interpretation.

Hence, if we could find S_2 effectively, we would have converted an unsolvable problem into a solvable one -- a contradiction.

□

2.1.9.3 Proof of Theorem 2.2 (Fundamental theorem of Herbrand schemas)

For Herbrand schemas, the notions of (a) halting, (b) divergence, (c) equivalence, (d) inclusion, (e) isomorphism, and (f) freedom, for all interpretations, are equivalent to the same notions for the Herbrand interpretations.

Proof. (Informal) (a), (b), (f) These are immediate from the definition of Herbrand schemas.

(c) This follows directly from (d) below.

(d) The "only if" part is trivial. For the "if" part, assume it is false. Then $S_1 \leq S_2$ on $\mathcal{U}$, but there is some interpretation I such that S_1 halts on I and S_2 does not halt with the same value. Now,

consider the Herbrand interpretation H corresponding to I . As S_1 is a Herbrand schema, S_1 halts on H .

(i) If $Val(S_2,I)$ is undefined then so is $Val(S_2,H)$ as S_2 is a Herbrand schema, and hence $S_1 \not\leq S_2$ on $\mathcal{X}$ -- a contradiction.

(ii) S_2 halts on I , and hence it also halts on H , and $Val(S_1,H) = Val(S_2,H)$, but $Val(S_1,I) \neq Val(S_2,I)$. We show that this is impossible by considering the (natural) homomorphism $\theta : H \to I$ from H onto the reachable elements in I (i.e., elements that can be expressed in constant terms). Then, we see by induction on the number of steps in the computation that at each step the values of variables in the computations of S_1 on H and I correspond with respect to θ ("variables" includes arrays, stacks, queues, counters, etc., and recursion is also handled -- and θ is extended to be the identity function over elements, like integers, that are not in the domain of H), and similarly for S_2 . Then we have $\theta(Val(S_1,H)) = Val(S_1,I)$, and $\theta(Val(S_2,H)) = Val(S_2,I)$, but $Val(S_1,H) = Val(S_2,H)$, and hence $Val(S_1,I) = Val(S_2,I)$ -- a contradiction.

(e) The "only if" part is trivial, and the "if" part follows on lines very similar to the proof of inclusion: if it is false then there must be a counterexample, say for an interpretation I , and $Seq(S_1,I)$ and $Seq(S_2,I)$ do not agree after some finite number of steps, but $Seq(S_1,H) = Seq(S_2,H)$ and values of variables correspond at each step for computations on I and H -- which yields a contradiction.

2.1.9.4 Proof of Theorem 2.3 (Schemas without equality are Herbrand)

Schemas that have no equality tests are Herbrand schemas.

Proof. (Informal) Assume the theorem is false. Then there is a schema S and an interpretation I for S (let the corresponding Herbrand interpretation be H) such that the paths of the computations of S on I and on H are different. Then they must first be different after a finite number of steps k . Then as in the proof of Theorem 2.2 (d), the values of variables in the two computations correspond for k-1 steps, and the k-th step must be a predicate test (since it must be a test, and tests on booleans yield the same value, and tests of equality are forbidden). But the outcome of the predicate test must be the same in both computations (by the definition of H corresponding to I) -- a contradiction.

□

2.1.9.5 Proof of Theorem 2.4 (value languages are r.e.)

The value language of any schema S is recursive enumerable.

It is easy to see that given any finite path in S (starting from the start statement) it is decidable whether or not the computation of S on some Herbrand interpretation follows this path. Also, given any path from the start statement to a halt statement, the output (for Herbrand interpretations) is fixed by the path, that is, if H_1, H_2 are two Herbrand interpretations on which the computations of S traverse the same path, then $Val(S,H_1) = Val(S,H_2)$.

We can now construct a partial recursive function from integers to strings whose range is precisely the value language of S :

"Let n be the input. Generate the n-th finite path in S (by any predefined ordering) and if it ends at a halt statement and can be traversed by some Herbrand interpretation, then output [Val(S,H)] where H is any such interpretation; otherwise diverge."

This completes the proof. □

2.2 Value Languages of Schemas

In this section, all schemas are assumed to have only monadic functions (zero-ary and unary) and arbitrary n-ary predicates, unless otherwise stated.

2.2.1 Flowchart Schemas

Theorem 2.7

The value languages of flowchart schemas (with monadic functions) that are free on the Herbrand interpretations are precisely the regular sets.

As a corollary, the value languages of free flowchart schemas with monadic functions and no equality, are regular (see Theorems 2.3 and 2.2f).

The proof is given in Section 2.2.3. It can be shown that the class of one-variable flowchart schemas (even with resets $y \leftarrow a_i$ and boolean variables, but without equality) can be translated to equivalent free schemas without equality, but with several variables. Then, from the proof of the above theorem and the Corollary 2.6 we have

Theorem 2.8. The value languages of schemas of $\mathcal{C}$(1 var) with monadic functions, are the regular sets.

The fact that all regular sets can in fact be generated is implicit in the proof given for the previous theorem in Section 2.2.3.

From Theorem 2.7 it follows that the following schema S_a is an inherently non-free schema, that is, it cannot be translated into an equivalent free flowchart schema (without equality tests).

S_a: START $\langle y_1, y_2 \rangle \leftarrow \langle a, a \rangle$;
<u>while</u> $p(y_1)$ <u>do</u> $y \leftarrow f(y_1)$;
<u>while</u> $p(y_2)$ <u>do</u> <u>begin</u> $y_1 \leftarrow g(y_1)$; $y_2 \leftarrow f(y_2)$ <u>end</u>;
HALT(y_1) .

The schema S_a is inherently non-free because $L(S_a) = \{g^n f^n \mid n \geq 0\}$ which is not a regular language. Note that the comment after Theorem 2.1 is implicitly used here in the unstated assumption that any equivalent free schema must have only monadic functions. However, S_a is indeed equivalent to a free recursive schema, and S_b is an example.

S_b: $F_0 \Leftarrow F_1(a)$;
$F_1(y) \Leftarrow$ <u>if</u> $p(y)$ <u>then</u> $gF_1f(y)$ <u>else</u> y .

The Theorems 2.7 and 2.8 do not apply to nonmonadic functions. As an example, consider the schema S_c .

S_c: START $y \leftarrow a$;
<u>while</u> $p(y)$ <u>do</u> $y \leftarrow f(y,y)$;
HALT(y) .

It has one variable, and it is free, but the value language $L(S_c)$ is $\{f^{2^n-1} \mid n \geq 0\}$, which is not even context free.

Theorem 2.9

The value languages of monadic schemas of $\mathcal{C}(2\ \text{var})$ are the recursively enumerable sets.

This is a slight generalization of a similar theorem due to Garland and Luckham [1971], in which they show that the value languages of monadic schemas of $\mathcal{C}()$ are the r.e. sets.

2.2.2 Recursive Schemas

Theorem 2.10

The value languages of recursive schemas (with monadic functions) that are free on the Herbrand interpretations are precisely the context free languages.

As a corollary, the value languages of free recursive schemas with monadic functions and no equality, are context free.

The proof can be found in Section 2.2.3. It follows from this that although the schema S_a in the previous section could be translated into an equivalent free recursive schema (S_b), the schema S_d cannot, for its value language is $\{f^n g^n f^n \mid n \geq 0\}$ which is not context free.

S_d: START $\langle y_1, y_2 \rangle \leftarrow \langle a, a \rangle$;
<u>while</u> $p(y_1)$ <u>do</u> $y_1 \leftarrow f(y_1)$;
<u>while</u> $p(y_2)$ <u>do</u> <u>begin</u> $y_1 \leftarrow g(y_1)$; $y_2 \leftarrow f(y_2)$ <u>end</u>;
$y_2 \leftarrow a$;
<u>while</u> $p(y_2)$ <u>do</u> <u>begin</u> $y_1 \leftarrow f(y_1)$; $y_2 \leftarrow f(y_2)$ <u>end</u>;
HALT(y_1) .

Theorem 2.11

The value languages of schemas of $\mathcal{C}(R, 1\text{var})$ with monadic functions, no resets, and no defined function inside atomic terms, are the context free languages.

Note: an atomic term α is a predicate or equality term used in a test (if α then ... else ...) or as a boolean argument. If in any function definition $F_i \Leftarrow$ if α then $\langle \tau, \alpha_1, \alpha_2, \ldots \rangle$ else $\langle \tau', \alpha_1', \alpha_2', \ldots \rangle$, $i \neq 0$, the terms τ or τ' contain a zero-ary function a_j, we call this a reset.

This theorem is a generalization of a similar theorem by Garland and Luckham [1971], and the proof is presented in Section 2.2.3. This theorem does not follow from Theorem 2.10 (as did Theorem 2.8 from Theorem 2.7) because there exist one-variable recursive schemas that cannot be made free. The following example, schema S_e, is due to Ashcroft, Manna, and Pnueli [1971].

S_e: $F_0 \Leftarrow F_1(a)$;
$F_1(y) \Leftarrow$ if $p(y)$ then $F_2F_1f(y)$ else y;
$F_2(y) \Leftarrow$ if $q(y)$ then $f(y)$ else y .

The theorem shows that the schema S_d, for example, cannot be translated into a recursive schema with one variable (and satisfying the conditions of the Theorem 2.11).

From the general result of McCarthy [1962] that any schema in $\mathcal{C}$(n var) can be effectively translated into an equivalent schema of $\mathcal{C}$(R,n var), and using the Theorem 2.4 we have the following.

Corollary 2.12. The value languages of monadic schemas of $\mathcal{C}$(R, 2 var) are the recursively enumerable sets.

2.2.3 Proofs of Theorems on Value Languages

2.2.3.1 Proof of Theorem 2.7

The theorem states that the value languages of flowchart schemas (with monadic functions) that are free on the Herbrand interpretations, are precisely the regular sets.

(1) We first show the easy part, that is, that all regular sets can be generated. Given any regular set over $\Sigma = \{f_1, f_2, \ldots, f_r\}$, we consider the deterministic finite state automaton A that accepts the regular set. Let its states be $Q = \{q_0, q_1, \ldots, q_k\}$ where q_0 is the start state, and $F \subset Q$ is the set of final states, and the next-state function is $\delta : Q \times \Sigma \to Q$. We construct a flowchart schema S with one variable which uses unary functions symbols $f_1, f_2, \ldots, f_r$, a zero-ary function symbol a, and unary predicates $p_1, p_2, \ldots, p_r$ (note: it would suffice to use $\log_2(r)$ predicates, but this construction is slightly simpler). We label statements $L_0, L_1, \ldots, L_k$ in correspondence with the states $q_0, q_1, \ldots, q_k$ of A. In addition there is one halt statement L : HALT(y). The start statement in S is:

START y ← a; goto L_0 .

Let $d_{i,j}$ denote $\delta(q_i, f_j)$. Then for $0 \leq i \leq k$ the statements of the schema are: if $q_i \in F$ (i.e., a final state) then

L_i : if $p_1(y)$ then begin $y \leftarrow f_1(y)$; goto $L_{d_{i,1}}$ end

else if $p_2(y)$ then begin $y \leftarrow f_2(y)$; goto $L_{d_{i,2}}$ end

⋮

else if $p_r(y)$ then begin $y \leftarrow f_r(y)$; goto $L_{d_{i,r}}$ end

else goto L

and if $q_i \notin F$ then

L_i : ⋮

else if $p_r(y)$ then begin $y \leftarrow f_r(y)$; goto $L_{d_{i,r}}$ end

else LOOP .

Now, the schema S is clearly free, and the computation can reach any statement L_i with value x (in the Herbrand domain) if and only if the string $[x]$ takes the automaton from the start state to state q_i (recall that $[x]$ denotes the string x with parentheses and unary function symbols removed). Thus the value language for S equals the given regular set.

(2) We now show that the value language of a free flowchart schema (with monadic functions) is regular.

Let S be a free flowchart schema, with variables $y_1, y_2, \ldots, y_n$, and unary functions $\Sigma = \{f_1, f_2, \ldots, f_r\}$. Without loss of generality we assume S has a single halt statement: HALT(y_1). We label the start statement, and all the assignment statements of S by $L_1, L_2, \ldots, L_k$.

Let $X_{i,j}$ denote the set of strings in Σ^* corresponding to the possible values of the variable y_j after statement L_i is executed (on a Herbrand interpretation). In addition, let X denote the set of strings corresponding to the possible outputs -- in other words, X is the value language.

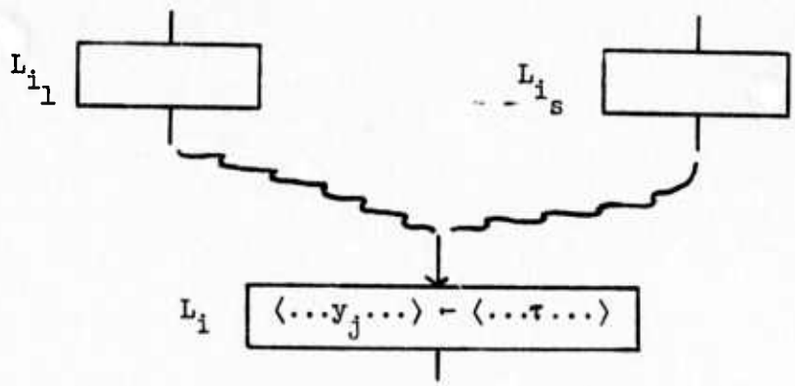
We will now demonstrate a set of recursive equations relating the X's and having the property that

(a) the least fixed point exists, and is regular, and

(b) the least fixed point corresponds to the values of the X's for the computations.

$X_{i,j}$: suppose $\{L_{i_1}, L_{i_2}, \ldots, L_{i_s}\}$ are the statements of S for which there is a path from L_{i_j} to L_i without passing through any assignment

statement. Now, each of the L_i's is an assignment statement of the form $\langle y_1,\ldots,y_n\rangle \leftarrow \langle \tau_1,\ldots,\tau_n\rangle$. Let the term in L_i for y_j be τ $(= \tau_j)$.



If τ is a constant term, we use the equation

$$X_{i,j} <= [\tau] \quad .$$

If, on the other hand, τ is a sequence of functions applied to one of the variables, say y_m, then we use

$$X_{i,j} <= [\tau]X_{i_1,m} + [\tau]X_{i_2,m} + \ldots + [\tau]X_{i_s,m}$$

where $+$ stands for union, and $[\tau]$ is the notion introduced earlier, but extended to nonconstant terms as well ($[\tau]$ is the string τ, omitting all parentheses, zero-ary functions, and variables). Note that the start statement is treated just like any other assignment statement.

X: Suppose $\{L_{i_1},\ldots,L_{i_s}\}$ are the statements of S for which there is a path to the single halt statement of S $(\text{HALT}(y_1))$. Then, we use

$$X <= X_{i_1,1} + X_{i_2,1} + \ldots + X_{i_r,1} \quad .$$

(a) We have a right-linear set of recursion equations (on strings), and such a system has a unique least fixed point, namely, the regular sets (one for each of the X's) obtained by these equations treated as productions.

(b) For convenience, we will rename $X, X_{1,1}, \ldots, X_{k,n}$ to be $Y_1, Y_2, \ldots, Y_m$; and we define the sets

$$Y_{1,0} = Y_{2,0} = \ldots = Y_{m,0} = \Phi$$

and

$$Y_{i,c+1} = \mathcal{F}_i(Y_{1,c}, Y_{2,c}, \ldots, Y_{m,c})$$

where $\mathcal{F}_i$ is the function used in the recursive definition (for Y_i):

$$Y_i \Leftarrow \mathcal{F}_i(Y_1, Y_2, \ldots, Y_m) \quad .$$

Then, the least fix-point $\bar{Y}_i$ for Y_i is given by

$$\bar{Y}_i = \bigcup_{c<\infty} Y_{i,c} = \text{the least fix-point} \quad . \qquad (*)$$

We define $Z_{i,c}$ to be the set of strings corresponding to the variable Y_i (which is some $X_{i,j}$ or X itself) obtained in not more than c steps of the computations of the schema S (for all Herbrand interpretations) where a "step" is defined to be the execution of the start statement, an assignment statement, or a halt statement (i.e., not loops or predicate tests). By definition,

$$\bigcup_{c<\infty} Z_{i,c}$$

is the set of strings corresponding to the variable Y_i in all possible computations. We have to show that

$$\bar{Y}_1 = \bigcup_{c<\infty} Z_{1,c} \quad ,$$

but for the induction to work we will prove the stronger result, that

$$\bar{Y}_i = \bigcup_{c<\infty} Z_{i,c} \quad \text{for all } i \leq m \ .$$

(i) To show that $\bar{Y}_i \subset \bigcup_{c<\infty} Z_{i,c}$

We will prove that $Y_{i,c} \subset Z_{i,k+c}$, and then by equation (*) the result follows.

The start step is trivial as $Y_{i,0} = \varphi$. For the induction step, $c \geq 0$, assume it is true for c, to show it for $c+1$.

Case 1. $Y_i <= x$ (where $x = [\tau]$ is a constant) is the recursion equation for Y_i. Now, as the schema is assumed to be free, and all statements are reachable, and there are only k start and assignment statements, the statement corresponding to Y_i must be executed within $k+1$ steps, i.e., $Z_{i,k+c+1} = \{x\}$, and, of course, $Y_{i,c+1} = \{x\}$, and hence $Y_{i,c+1} \subset Z_{i,k+c+1}$.

Case 2. $Y_i <= xY_{i_1} + \ldots + xY_{i_s}$, $(x = [\tau])$, where the statements corresponding to $Y_{i_1}, \ldots, Y_{i_s}$ lead to the statement for Y_i without any intervening assignment (or halt) -- note: only Y_1 corresponds to the halt statement. Since the schema is free, all paths can be taken, and by the definitions of $Y_{i,c}$, and $Z_{i,c}$ we have

$$\begin{aligned} Y_{i,c+1} &= xY_{i_1,c} + \ldots + xY_{i_s,c} && \text{(def)} \\ &\subset xZ_{i_1,k+c} + \ldots + xZ_{i_s,k+c} && \text{(ind hyp)} \\ &= Z_{i,k+c+1} \ . && \text{(def)} \end{aligned}$$

(ii) To show that $\bigcup_{c<\infty} Z_{i,c} \subset \bar{Y}_i$

We will prove that $Z_{i,c} \subset Y_{i,c}$.

The start step is trivial, for after zero steps of the computation, all $Z_{i,0}$'s are φ . For the induction step, $c \geq 0$, assume the result is true for c , to prove it for $c+1$:

Case 1. If $y_i <= x$, then $Y_{i,c+1} = \{x\}$, and $Z_{i,c+1}$ can only be φ or $\{x\}$.

Case 2. If $Y_i <= xY_{i_1} + \ldots + xY_{i_s}$ then, as before,

$$Z_{i,c+1} = xZ_{i_1,c} + \ldots + xZ_{i_s,c} \qquad \text{(def)}$$

$$\subset xY_{i_1,c} + \ldots + xY_{i_s,c} \qquad \text{(ind hyp)}$$

$$= Y_{i,c+1} \ . \qquad \text{(def)}$$

This completes the proof of Theorem 2.7.

□

2.2.3.2 Proof of Theorem 2.9

The value languages of monadic schemas of $\mathcal{C}$(2 var) are the recursively enumerable sets.

We use the fact that a recursively enumerable set is generated by the outputs of a Turing machine, and that all r.e. sets can be so generated. Luckham, Park, and Paterson [1970] have shown how a two-variable schema S using a unary function f and a unary predicate p can simulate a Turing machine computation such that S diverges unless

the Turing machine halts, and if the machine halts then its output can be "read off" by the values $p(y_1), p(f(y_1)), p(ff(y_1)), p(fff(y_1)), \ldots$ in some coded form, where y_1 is one of the variables of the schema. We modify the schema S so that before halting it resets y_2 to a $(y_2 \leftarrow a)$, and then proceeds to apply the appropriate functions to y_2 as read off by the variable y_1, and then halts, outputing y_2. We thus obtain a subclass of $\mathcal{C}$(2 var) whose value languages are the r.e. sets, thus proving the theorem by recourse to the Theorem 2.4 that the value language of any schema in $\mathcal{C}$(2 var) is r.e.

□

2.2.3.3 Proof of Theorem 2.10

The value languages of the recursive schemas (with monadic functions) that are free on the Herbrand interpretations are the context free languages.

(1) We will first prove the simpler part, that is, that all context free languages can be generated.

Let G be any context free grammar over the nonterminals $F_1, F_2, \ldots$, and the terminals $f_1, f_2, \ldots$, where F_1 is the start symbol. We assume G is in Greibach normal form, that is, all productions have the form

$$F_i \rightarrow F_{i_1} F_{i_2} \cdots F_{i_k} f_j \quad .$$

Suppose there are at most m productions for any F_i, then in our schema we will have m-1 unary predicates $p_1, p_2, \ldots, p_{m-1}$. In the schema we will allow definitions like (a) $F_i(y) \Leftarrow \tau$, and also (b) nested if-then-else's, with the understanding that these features

are easily eliminated by (a) substituting, and (b) adding new defined functions, without destroying the property of freedom in our particular construction. The schema is:

$$F_0 \Leftarrow F_1(a) ,$$

and for each F_i in G , if there are n -productions for F_i :

$$F_i \rightarrow F_{1,1}\, F_{1,2} \cdots F_{1,k_1}\, f_{j_1}$$
$$\vdots$$
$$F_i \rightarrow F_{n,1}\, F_{n,2} \cdots F_{n,k_n}\, f_{j_n}$$

then the corresponding defined function in S is:

$$F_i(y) \Leftarrow \underline{\text{if}}\ p_1(y)\ \underline{\text{then}}\ F_{1,1}(F_{1,2} \cdots F_{1,k_1}(f_{j_1}(y)))$$
$$\underline{\text{else}}$$
$$\vdots$$
$$\underline{\text{else if}}\ p_{n-1}(y)\ \underline{\text{then}}\ F_{n-1,1}(\cdots(f_{j_{n-1}}(y)))$$
$$\underline{\text{else}}\ F_{n,1}(\cdots(f_{j_n}(y))) \quad .$$

It is easy to see that this schema is free, and its value language equals the language generated by the grammar G .

(2) We now prove that the value language of any free recursive schema is context free.

Given a free recursive schema S using only monadic base functions, we construct a context free grammar G such that the value language of S is the same as the language generated by G , S has the form

S: $F_0 \Leftarrow \tau_0()$;

$F_1(\bar{y},\bar{z}) \Leftarrow \underline{\text{if}}\ \alpha_1\ \underline{\text{then}}\ \bar{\tau}_1\ \underline{\text{else}}\ \bar{\tau}_1'$

$\vdots$

$F_k(\bar{y},\bar{z}) \Leftarrow \underline{\text{if}}\ \alpha_k\ \underline{\text{then}}\ \bar{\tau}_k\ \underline{\text{else}}\ \bar{\tau}_k'$.

We will assume that no short-cut notation is used; for example, if F returns just one data value, to obtain it we must write $Y_1(F(\ldots))$ instead of just $F(\ldots)$. Similarly, if F_1 returns a vector that matches the arguments for F_2 , we must write $F_2(Y_1(F_1(\ldots)),Y_2(F_1(\ldots)),\ldots)$ instead of $F_2(F_1(\ldots))$.

The terminals of the grammar G to be generated are the unary function symbols of S . The nonterminals have the form

$$(Y_i,F_j,y_k)$$

which has the following significance: if the defined function F_j is entered with any string x for its k-th data argument, then (Y_i,F_j,y_k) represents the possible strings x' that could have been added to the left of x such that the i-th data argument of F_j can exit with this value (i.e., $x'.x$). The other type of nonterminal is

$$(Y_i,F_j)$$

which represents the strings $Y_i(F_j(\ldots))$ could exit with no matter what the arguments.

To construct G , we first define the following notation:

$$[\tau]_{y_i}$$

where τ is any term (which may use the defined functions) to be a set of strings as follows:

(1) for any zero-ary function a : $[a]_{y_i} = \varphi$

(2) for any y_i : $[y_i]_{y_i} = \Lambda$ */

and for $j \neq i$ $[y_j]_{y_i} = \varphi$

(3) for any unary f : $[f(\tau)]_{y_i} = f.[\tau]_{y_i}$

(4) and $[Y_j(F(\tau_1,\tau_2,\ldots))]_{y_i} = \bigcup_k (Y_j,F,y_k).[\tau_k]_{y_i}$

for all k varying over the data arguments of F .

And similarly, $[\tau]_0$ is defined as follows:

(1) for any zero-ary function a : $[a]_0 = \Lambda$

(2) for any y_i : $[y_i]_0 = \varphi$

(3) for any f : $[f(\tau)]_0 = f.[\tau]_0$

(4) and $[Y_j(F(\tau_1,\tau_2,\ldots))]_0 = (Y_j,F) + \bigcup_k (Y_j,F,y_k).[\tau_k]_0$

Note: we are using both the signs $\cup$ and + (for strings) to mean union.

As an example

$$[Y_2(F(fg(a),Y_1(G(y_3,y_1,a,fy_3)),hy_3))]_{y_3}$$

$$= (Y_2,F,y_2)(Y_1,G,y_1)$$

$$+ (Y_2,F,y_2)(Y_1,G,y_4)f$$

$$+ (Y_2,F,y_3)h$$

and

*/ Note that the notation is a little informal. We should strictly write $[y_i]_{y_i} = \{\Lambda\}$, etc.

$$[Y_2(F(fg(a),Y_1(G(y_3,y_1,a,fy_3)),hy_3))]_0$$

$$= (Y_2,F)$$

$$+ (Y_2,F,y_1)\,f\,g$$

$$+ (Y_2,F,y_2)(Y_1,G)$$

$$+ (Y_2,F,y_2)(Y_1,G,y_3) \quad .$$

Given the free schema, we can separate the defined functions into two classes -- those that can eventually return, and those that must diverge. This can be done by building up the set of functions that can halt; starting with the null set:

$$F(\ldots) \Leftarrow \underline{\text{if}}\ \alpha\ \underline{\text{then}}\ \bar{\tau}\ \underline{\text{else}}\ \bar{\tau}' \quad .$$

F can halt if α can halt (i.e., all defined functions in it can halt) and so can one of $\bar{\tau}$ or $\bar{\tau}'$.

The construction of the grammar G ignores all boolean variables, all tests, and all defined functions that must diverge. If the start function F_0 diverges, then the language is the empty set. Otherwise, we build G as follows:

(1) $\quad F_0 \Leftarrow \tau() \quad .$

The start nonterminal in G is (Y_1,F_0) :

$$(Y_1,F_0) \rightarrow [\tau()]_0 \quad .$$

(2) $\quad F_i(y_1,y_2,\ldots,z_1,z_2,\ldots) \Leftarrow \underline{\text{if}}\ \alpha\ \underline{\text{then}}\ \bar{\tau}\ \underline{\text{else}}\ \bar{\tau}'$

where F_i is a function that can halt (which implies that α can halt). Then, for all Y_j,y_k (that make sense for F_i), if $\bar{\tau} = \langle\tau_1,\tau_2,\ldots\rangle$,

and it can halt, then

$$(Y_j,F_i,y_k) \rightarrow [\tau_j]_{y_k}$$

and

$$(Y_j,F_i) \rightarrow [\tau_j]_0$$

and similarly for $\bar{\tau}'$.

We can show that G generates the value language of S on lines similar to the proof of Theorem 2.7. We consider a Herbrand interpretation over the given base functions and predicates and also over a special set of zero-ary functions $b_1,b_2,\ldots,b_n$ where n is the number of variables in S . Then, for any F_i and integer c , we associate the sets $(Y_j,F_i,y_k)'_c$ which stand for the possible strings x , such that if $F_i(y_1,y_2,\ldots,z_1,z_2,\ldots)$ is entered with $y_1 = b_1$, $y_2 = b_2$, ... , then $Y_j(F_i(\ldots))$ exits with value $x.b_k$ (for all possible values of the z_i's) without executing recursive calls of depth more than c . And similarly, $(Y_j,F_i)'_c$ stands for the strings x such that $Y_j(F_i(\ldots))$ exits with value $x.a_k$ (for any k , and the same arguments to F_i as before). Note: by the depth of recursive calls we do not include recursive calls required to evaluate any test α in $F_i \Leftarrow \underline{\text{if}}\ \alpha\ \underline{\text{then}}\ \bar{\tau}\ \underline{\text{else}}\ \bar{\tau}'$. We can then show by least fixed-point arguments that

$$\bigcup_{c<\infty} (Y_j,F_i,y_k)'_c = L_G(Y_j,F_i,y_k)$$

where the right hand side represents the strings generated by the nonterminal (Y_j,F_i,y_k) in the grammar G ; and similarly for (Y_j,F_i) . Thus $L_G(Y_1,F_0)$ does represent the possible output strings in this

augmented Herbrand interpretation (with the additional zero-ary functions $b_1, b_2, \ldots$). But the computation for I_0 never computes any element $x.b_i$, and hence the possible output strings are the same for unaugmented Herbrand interpretations (without the $b_1, b_2, \ldots$). □

2.2.3.4 Proof of Theorem 2.11

It is easy to see that all context free languages are generated by one-variable monadic recursive schemas without resets. The construction in the previous section applies.

To show that only context free languages are generated, let S be a given one-variable recursive schema such that no atomic term has a defined function, and S has no resets. We define the depth $|\tau|$ of a term τ (constant or variable) to be the depth of nesting of function symbols $|a_i| = |y_i| = 0$, $|f(\tau_1, \ldots, \tau_n)| = \max(|\tau_1|, \ldots, |\tau_n|)+1$. Let k be the largest depth of any term used in S. A specification state Q of S defines all predicates on all terms $\tau()$ and $\tau(y)$ such that $|\tau()|, |\tau(y)| \leq k$. In addition, it may also specify $y = \tau()$ for some $\tau()$ with $|\tau()| \leq k$ -- in which case the values of predicates respect this specification. Now, given the specification state Q for y, it is clear how it may be updated, i.e., we can determine all possible Q' for $f(y)$ (for any unary function f). Note that the updating is done only for the Herbrand interpretations. Also note that n-ary predicate symbols and equality tests are handled by this mechanism.

Without loss of generality we can assume that in S, no defined function is passed any boolean arguments -- any schema S can be translated into this form by creating many copies of each defined function, and testing all boolean arguments of the (old) function before the (new) function is called (this yields nested if-then-else's which can then

be eliminated). Then, as the schema cannot test any booleans returned by functions, we can simply remove them and get an equivalent schema that uses no booleans at all.

Now, from the schema S we construct a context free grammar G as follows. The nonterminals are of the form

$$(Q', F_i, Q)$$

where Q, Q' are specification states, and there is a special start symbol: (F_0). Given a term τ and specification states Q', Q we define a set of strings (notation $Q'[\tau]_Q$) of terminals and nonterminals of G as follows:

(1) $Q'[a_i]_Q$ is Λ if the predicates over constants agree on Q and Q', and in Q', $y = a_i$ is specified; otherwise $Q'[\tau]_Q$ is φ.

(2) $Q'[y]_Q$ is Λ if $Q' = Q$; otherwise it is φ.

(3) $Q'[f_i(\tau)]_Q$ is $\cup\, f_i . Q''[\tau]_Q$ where the union is taken over all Q'' that can be updated to Q' by applying f_i.

(4) $Q'[F_i(\tau)]_Q$ is $\cup\, (Q', F_i, Q'') . Q''[\tau]_Q$ for all Q''.

We can now define the grammar G.

(1) $F_0 \Leftarrow \tau$ is converted into the following productions for the start symbol (F_0) of G:

$$(F_0) \rightarrow Q'[\tau]_Q$$

for all Q', Q.

(2) $F_i(y) \Leftarrow \underline{\text{if}}\ \tau_1 = \tau_2\ \underline{\text{then}}\ \tau\ \underline{\text{else}}\ \tau'$.

For all Q in which the terms τ_1 and τ_2 are equal (note:

τ_1, τ_2 do not use any F_i)

$$(Q', F_i, Q) \rightarrow Q'[\tau]_Q$$

for all Q' , and for all other Q :

$$(Q', F_i, Q) \rightarrow Q'[\tau']_Q \quad .$$

(3) $F_i(y) \Leftarrow$ if $p_j(\tau_1, \tau_2, \ldots)$ then τ else τ' .

For all Q in which $p_j(\tau_1, \tau_2, \ldots)$ is true:

$$(Q', F_i, Q) \rightarrow Q'[\tau]_Q$$

and for all other Q :

$$(Q', F_i, Q) \rightarrow Q'[\tau']_Q \quad .$$

□

This lemma includes the following simple generalizations over a similar result of Garland and Luckham: (1) boolean variables, (2) tests on constant terms and terms using the variable y , (3) equality tests, and (4) n-ary predicates.

2.3 The Power of Classes of Schemas

2.3.1 On the Number of Variables in Schemas

It is evident that any flowchart schema S which uses n boolean variables can be translated into an isomorphic (and hence equivalent) flowchart schema with no boolean variables. This can be accomplished by creating at most 2^n "copies" of S , one copy for each possible set of values for the n boolean variables.

Similarly, any recursive schema can be translated into an equivalent recursive schema in which no argument of any defined function is a boolean variable. We now wish to show that the same is true for the values returned by the defined functions as well. In fact, we will show a stronger result: that any recursive schema S_1 can be translated into an equivalent recursive schema S_2 which uses only data values, and each defined function returns just one value. It is possible, however, that the number of operations executed by S_2 may be an exponential of the operations of S_1 (for any interpretation).

Theorem 2.13. Every schema $S_1 \in \mathcal{C}(R)$ (or in $\mathcal{C}(R, =)$) can be effectively translated into an equivalent schema S_2 in the same class such that only data arguments are passed to each defined function in S_2, and each defined function returns exactly one data value (and no boolean values).

For the proof see Section 2.3.4.

Now that we have succeeded in restricting each defined function to returning just one value (while retaining the power of all recursive schemas), the natural question that arises is whether we can also restrict the number of arguments to be one, or if not, to two, or to some integer n. And a similar question may be asked for flowchart schemas. Value language considerations show, for example, that one-variable flowchart schemas cannot give us the power of all flowchart schemas -- the value languages are regular (for monadic functions), whereas for two-variable schemas the value languages are all the r.e. sets. The following theorem puts such speculation to rest.

Theorem 2.14

(a) $\mathcal{C}(0\text{ var}) \equiv \mathcal{C}(R,0\text{ var})$,

(b) $\mathcal{C}() \not\leq \mathcal{C}(R,1\text{ var})$, and

(c) $\mathcal{C}(n+1\text{ var}) \not\leq \mathcal{C}(R,n\text{ var})$ for $n \geq 0$.

Part (a) of this theorem is trivial.

Part (b) was shown by Paterson and Hewitt [1970] by showing that no flowchart schema is equivalent to the following recursive schema S_a (we use nested if-then-else's with the comment that they can be removed to obtain a strictly "legal" one-variable recursive schema):

S_a: $F_0 \Leftarrow F(a)$;

$F(y) \Leftarrow$ if $p(f_1(y))$ then if $p(f_2(y))$ then y
else $F(f_2(y))$
else if $p(F(f_1(y)))$ then $F(f_2(y))$
else a .

This schema checks to see if there is an infinite sequence $f_{i_1}, f_{i_2}, f_{i_3}, \ldots$, each $i_j = 1$ or 2 , such that all the tests $p(f_{i_1}(a)), p(f_{i_2} f_{i_1}(a)), p(f_{i_3} f_{i_2} f_{i_1}(a)), \ldots$ are false. The schema halts only if no such sequence exists.

Part (c) of this theorem can be shown by demonstrating that the following problem can be solved with an (n+1)-variable flowchart schema, but not with any n-variable recursive schema (without equality). The problem is:

" if there exist integers i,j , $0 \leq i \leq n$, $0 \leq j$ such that $p(g^j f^i(a))$ is false then halt (with output a), else diverge ".

For details, see Section 2.3.4.

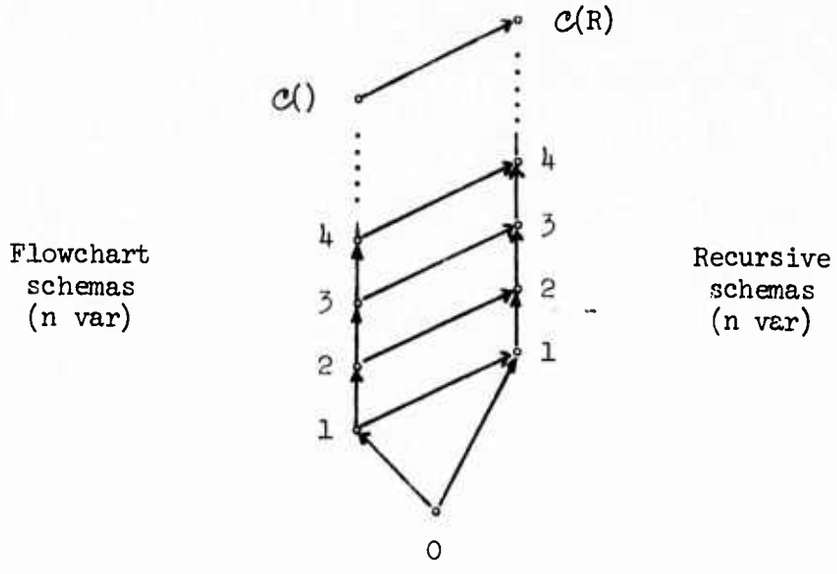


Figure 2.1

The consequence of this theorem is that we can draw the diagram relating flowchart and recursive schemas. In Figure 2.1 an arrow $A \rightarrow B$ indicates the relation " B is strictly more powerful than A ".

2.3.2 Equality Tests

A problem is said to be a Herbrand problem if it can be solved by some Herbrand schema. Otherwise, if it can only be solved by an inherently non-Herbrand schema it is called a non-Herbrand problem. All schemas in $\mathcal{C}(pds,q,list,A)$, $\mathcal{C}(R)$ are Herbrand schemas, and none of them can solve any non-Herbrand problem. However, there exist some very simple non-Herbrand problems which can be solved by schemas in $\mathcal{C}(=)$, for example, given two zero-ary functions a_1, a_2 the problem

P_a = " if $a_1 = a_2$ then halt (with output a_1), otherwise diverge "

can be solved by the schema

START $y \leftarrow a_1$;

<u>if</u> $a_1 = a_2$ <u>then</u> HALT(y) <u>else</u> LOOP ,

demonstrating that $\mathcal{C}(=) \not\leq \mathcal{C}(\text{pds,q,list,A})$, and $\mathcal{C}(=) \not\leq \mathcal{C}(R)$.

To demonstrate the power of equality tests we present two other (more interesting) non-Herbrand problems that can be solved by schemas in $\mathcal{C}(A, =)$.

<u>Example 1 -- Inverse of a Unary Function</u>

The problem is:

P_b = " given a unary function symbol f , a zero-ary function constant a , and a finite number of other n-ary function symbols, $n \geq 0$, write a program schema that under any interpretation will yield a value of "$f^{-1}(a)$" as output. That is, it should find an element y that can be expressed in terms of the given function symbols such that $f(y) = a$; and if no such element exists, the schema should diverge ".

This is a non-Herbrand problem because for no Herbrand interpretation does there exist an element y such that $f(y) = a$, and hence, if any Herbrand schema S claims to solve it, S diverges on all Herbrand interpretations, and hence on all interpretations (by Theorem 2.2) and this is certainly not the desired behavior. A schema that solves the problem is presented in Section 2.3.4.

Example 2 -- Herbrand-like Interpretations

Given a set of function and predicate symbols of which there is at least one zero-ary function, we say that an interpretation I for this set is Herbrand-like if there exists some Herbrand interpretation H such that there is a 1-1 homomorphism from H into I. In other words, an interpretation I is Herbrand-like if and only if for every pair of distinct terms τ_1 and τ_2 (made up of the given functions) the elements in I corresponding to t_1 and t_2 are distinct.

Now, consider the following problem:

P_c = " given an interpretation for a set of function and predicate symbols, of which at least one is a zero-ary function a, determine if the interpretation is not Herbrand-like. If the interpretation is not Herbrand-like then halt with output a, else diverge ".

This problem is inherently non-Herbrand in nature because a schema that solves this problem must diverge for every Herbrand interpretation. But for certain other interpretations the schema should halt. A schema with equality tests that solves the problem P_c is presented in Section 2.3.4.

The problem P_c is an abstract model closely related to certain problems in real life programming. As an illustration, consider a directed graph (with an identified root node) in which each node has two identified pointers leading from it. Pointers may lead to a terminal node "NIL". The problem is to determine whether or not the given graph is a tree. This problem may be modeled by the above problem with two monadic functions representing the two pointers, and with the difference

that the search for the equality of two "terms" is conducted not for the entire set of all terms, but for those terms not representing NIL. The correspondence is that the interpretation is "Herbrand-like" for this set of terms if and only if the corresponding graph is a tree. Another related problem is that of determining if a given LISP list is circular. Here, the two pointers from a node represent the car , and the cdr of the list represented by the node.

While equality tests are necessary to solve some non-Herbrand problems, equality can be used to solve Herbrand problems as well. We give two examples of Herbrand problem which are solved by schemas with equality.

Example 3 -- Expose the False One (or, the Witch Hunt)

The problem is

P_d = " if there exists an element x of the form $g^j f^i(a)$, $i,j \geq 0$, such that $p(x)$ is false, then halt (with output a), otherwise diverge ".

Our discussion on Theorem 2.14 indicates that no flowchart or recursive schema (without equality) can solve this problem. However, there is a non-herbrand schema in $\mathcal{C}(=)$ that can solve it -- see Section 2.3.4. And yet, it may be noted that P_d is a Herbrand problem for it can be solved by a schema in $\mathcal{C}(c)$.

Example 4 -- Translation of Flowchart Schemas with One Counter

The recursive schema

$F_0 \Leftarrow F(a)$;

$F(y) \Leftarrow$ if $p(y)$ then $f(y)$ else $F(G(f(y)))$;

$G(y) \Leftarrow$ if $q(y)$ then $g(y)$ else $G(G(g(y)))$,

is a canonical form for schemas in $\mathcal{C}(1c,=)$ in that any schema in $\mathcal{C}(1c,=)$ is equivalent to the above schema by giving appropriate meanings to a, f, g, p, q. (Note: these functions and predicates need not be total, but each can be implemented using only iteration.) This recursive schema can be translated into an equivalent schema from $\mathcal{C}(1c)$. Plaisted [1972] showed that it could also be translated into a rather large schema from $\mathcal{C}()$. However, the use of equality gives a simple schema equivalent to the recursive schema. And, in fact, this can be used as a basis to show that any schema in $\mathcal{C}(1c)$ or $\mathcal{C}(1c,=)$ can be converted quite easily into an equivalent schema in $\mathcal{C}(=)$. For details, see Section 2.3.4.

Now, the relations between classes of schemas with and without equality can be summed up as follows:

Theorem 2.15. $\mathcal{C}(\text{features}) < \mathcal{C}(\text{features}, =)$, where by "features" we mean such things as variables, counters, stacks, queues, lists, arrays, recursion, but excluding equality itself.

2.3.3 Counters, Stacks, Recursion, Arrays, etc.

In this section we wish to demonstrate the relationships between the various classes of schemas, and in particular we wish to show the partial ordering suggested by Figure 2.2.

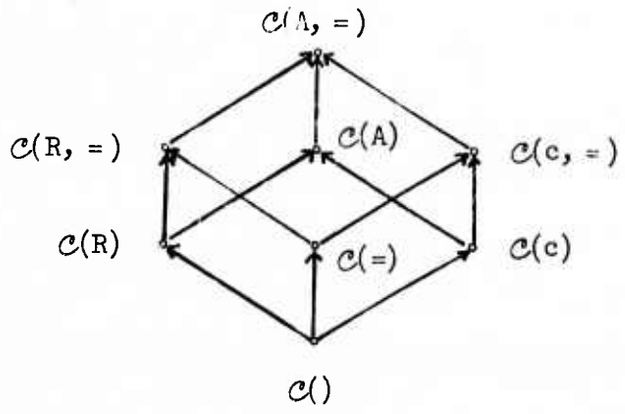


Figure 2.2. The power of schemas

In the figure, all arrows $A \rightarrow B$ indicate that " B is strictly more powerful than A ". Classes that cannot be linked by the transitive closure are indeed unrelated, for example, $\mathcal{C}(A) \not\leq \mathcal{C}(=)$, and $\mathcal{C}(=) \not\leq \mathcal{C}(A)$.

The following suffice to prove the relations shown in Figure 2.2 above.

Theorem 2.16

(1 - 4) $\mathcal{C}(R) \geq \mathcal{C}()$, $\mathcal{C}(R,=) \geq \mathcal{C}(=)$, $\mathcal{C}(A) \geq \mathcal{C}(c)$, $\mathcal{C}(A,=) \geq \mathcal{C}(c,=)$.

(5 - 8) $\mathcal{C}(c) \geq \mathcal{C}()$, $\mathcal{C}(c,=) \geq \mathcal{C}(=)$, $\mathcal{C}(A) \geq \mathcal{C}(R)$, $\mathcal{C}(A,=) \geq \mathcal{C}(R,=)$.

(9 - 12) $\mathcal{C}(=) \geq \mathcal{C}()$, $\mathcal{C}(R,=) \geq \mathcal{C}(R)$, $\mathcal{C}(c,=) \geq \mathcal{C}(c)$, $\mathcal{C}(A,=) \geq \mathcal{C}(A)$.

(13 - 15) $\mathcal{C}(A) \not\geq \mathcal{C}(=)$, $\mathcal{C}(R,=) \not\geq \mathcal{C}(c)$, $\mathcal{C}(c,=) \not\geq \mathcal{C}(R)$.

Of these, (3)-(6) and (9) - (12) are immediate, (1) and (2) have been known for a long time -- see McCarthy [1962], and (7), (8) follow easily from a similar result due to Constable and Gries [1972] and using Theorem 2.13. Part (13) is immediate because schemas in $\mathcal{C}(=)$ can solve non-Herbrand problems (e.g. P_a in Section 2.3.2) and these cannot be solved by schemas in $\mathcal{C}(A)$. For proofs of (14), (15), see Section 2.3.4.

Theorem 2.17 (One-counter Theorem)

(a) $\mathcal{C}() \equiv \mathcal{C}(1c)$, and

(b) $\mathcal{C}(=) \equiv \mathcal{C}(1c,=)$.

This was proved by Plaisted [1972]. Intuitively, the reasoning is that given a one-counter schema, one can get rid of the counter and replace it with a few variables which can then simulate the counter by "counting" on the interpretation itself, that is, on the values taken on by the other variables of the schema along the path of the computation.

Theorem 2.18 (Two-counter Theorem)

(a) $\mathcal{C}(c) \equiv \mathcal{C}(2c)$, and

(b) $\mathcal{C}(c,=) \equiv \mathcal{C}(2c,=)$.

To see that $\mathcal{C}(c) \equiv \mathcal{C}(2c)$, and $\mathcal{C}(c,=) \equiv \mathcal{C}(2c,=)$, observe that two counters are adequate for simulating the behavior of n counters for any n (Hopcroft and Ullman [1969], pg. 100) as follows: let $c_1', c_2', \ldots, c_n'$ be the n counters, and c_1 , c_2 be the two that are to simulate them -- the value of c_1 is to be $2^{c_1'} 3^{c_2'} 5^{c_3'} 7^{c_4'} \ldots \pi_n^{c_n'}$ where π_n is the n-th prime number: then, incrementing c_i' is like multiplying c_1 by π_i , decrementing c_i' is like dividing c_1 by π_i , and testing c_i' for zero is like testing if π_i divides c_1 -- all these operations can be performed by using c_2 to temporarily store an integer.

Theorem 2.19 (Recursion vs. a Stack, and a List)

(a) $\mathcal{C}(R) \equiv \mathcal{C}(1 \text{ pds}) \equiv \mathcal{C}(1 \text{ list})$, and

(b) $\mathcal{C}(R,=) \equiv \mathcal{C}(1 \text{ pds},=) \equiv \mathcal{C}(1 \text{ list},=)$.

That a pushdown stack is at least as powerful as recursion is not unexpected -- the concept that recursion can be implemented by a stack has been around for a long time in the theory of compilers. The converse, that recursion is as powerful as a pushdown stack is perhaps not so obvious; but it is certainly not mysterious considering that in recursion we allow the defined functions to return a vector of arguments (see, however, Theorem 2.13). Relating stacks to lists, it is clear that a list can do anything a stack can. That one list is not (strictly) more

powerful than a stack is interesting, but is not of any overwhelming importance because this result seems to depend on the kind of basic statements list schemas are endowed with.

Our last theorem deals with the equivalences of a large number of classes of schemas, sometimes also called the "maximal" classes.

Theorem 2.20 (Maximal Classes of Schemas)

(a) $\mathcal{C}(\text{pds,q,list,A}) \equiv \mathcal{C}(\text{1 pds,lc}) \equiv \mathcal{C}(\text{2 pds}) \equiv \mathcal{C}(\text{1 list,lc})$

$\equiv \mathcal{C}(\text{2 list}) \equiv \mathcal{C}(\text{lq}) \equiv \mathcal{C}(\text{lA})$, and

(b) $\mathcal{C}(\text{pds,q,list,A,=}) \equiv \mathcal{C}(\text{1 pds,lc,=}) \equiv \mathcal{C}(\text{2 pds,=}) \equiv \mathcal{C}(\text{1 list,lc,=})$

$\equiv \mathcal{C}(\text{2 list,=}) \equiv \mathcal{C}(\text{lq,=}) \equiv \mathcal{C}(\text{lA,=})$.

To prove this theorem it suffices to prove

$\mathcal{C}(\text{pds,q,list,A}) \equiv \mathcal{C}(\text{1 pds,lc}) \equiv \mathcal{C}(\text{lq})$, and

$\mathcal{C}(\text{pds,q,list,A,=}) \equiv \mathcal{C}(\text{1 pds,lc,=}) \equiv \mathcal{C}(\text{lq,=})$

because a list is at least as powerful as a stack, and a stack is at least as powerful as a counter; and further, the operation of a stack can be simulated with an array (with counters to subscript it, of course). The proof is indicated in Section 2.3.4. Note that to use an array, at least one counter is required; and one counter is also sufficient in that the class of schemas in $\mathcal{C}(\text{lA})$ with just one counter is as powerful as $\mathcal{C}(\text{lA})$ itself, and similarly for $\mathcal{C}(\text{lA,=})$. We may also remark here that for schemas restricted to monadic functions, flowchart schemas augmented with two variables have all the power of the maximal classes, that is,

$\mathcal{C}(\text{2c,monadic fns}) \equiv \mathcal{C}(\text{pds,q,list,A,monadic fns})$, and

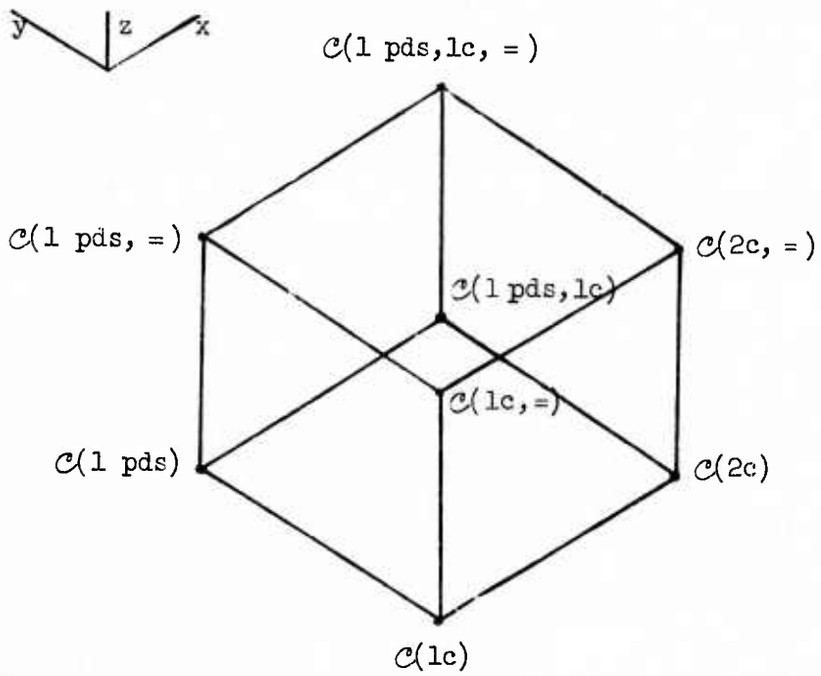
$\mathcal{C}(\text{2c,=,monadic fns}) \equiv \mathcal{C}(\text{pds,q,list,A,=,monadic fns})$.

It is interesting to label the vertices in Figure 2.2 in another way as shown in Figure 2.3. This figure can be treated as a unit cube where the axes are labelled as follows:

x-axis: "add a counter",

y-axis: "delete a counter, and add a stack", and

z-axis: "add equality tests".



$\mathcal{C}(1c) \equiv \mathcal{C}()$

$\mathcal{C}(1c, =) \equiv \mathcal{C}(=)$

$\mathcal{C}(2c) \equiv \mathcal{C}(c)$

$\mathcal{C}(2c, =) \equiv \mathcal{C}(c, =)$

$\mathcal{C}(1 \text{ pds}) \equiv \mathcal{C}(R) \equiv \mathcal{C}(1 \text{ list})$

$\mathcal{C}(1 \text{ pds}, =) \equiv \mathcal{C}(R, =) \equiv \mathcal{C}(1 \text{ list}, =)$

$\mathcal{C}(1 \text{ pds},1c) \equiv \mathcal{C}(A) \equiv \mathcal{C}(1 \text{ list},1c) \equiv \mathcal{C}(2 \text{ pds}) \equiv \mathcal{C}(2 \text{ list}) \equiv \mathcal{C}(1q)$

$\equiv \mathcal{C}(1A) \equiv \mathcal{C}(\text{pds},q,\text{list},A)$

$\mathcal{C}(1 \text{ pds},1c, =) \equiv \mathcal{C}(A, =) \equiv \mathcal{C}(1 \text{ list},1c, =) \equiv \mathcal{C}(2 \text{ pds}, =) \equiv \mathcal{C}(2 \text{ list}, =)$

$\equiv \mathcal{C}(1q, =) \equiv \mathcal{C}(1A, =) \equiv \mathcal{C}(\text{pds},q,\text{list},A, =)$

Figure 2.3

Note that the Figures 2.2 and 2.3 are "isomorphic".

Intuitively, there seem to be three inherent factors that determine the power of schemas.

(1) The amount of data space. Flowchart schemas, even with counters and equality tests have a fixed finite amount of space, that is, the number of data variables. It is for this reason that they cannot compute very large terms that require the saving of an arbitrarily large number of data values. For example, no schema in $\mathcal{C}(c, =)$ is equivalent to the recursive schema

$$F_0 \Leftarrow F_1(a);$$

$$F_1(y) \Leftarrow \underline{\text{if}}\ p(y)\ \underline{\text{then}}\ h(F(f(y)),F(g(y)))\ \underline{\text{else}}\ y \quad .$$

Recursive schemas act as if they have an unbounded amount of space, as do schemas with stacks, queues, lists or arrays. The amount of space available to a schema is, however, not a limitation when only schemas with monadic functions are considered since in that case any (constant) term can be computed with only one data variable by applying the proper base functions in the right order.

(2) The control capability. Boolean variables and counters are examples of control features. We have seen, however, that boolean variables add no inherent power (except to make a schema more compact). And two counters add as much control capability as one might want because we can simulate the computation of a Turing machine (with zero input). The question then is whether or not one counter adds any power. The answer is that it depends on the schema. For example, the addition of one counter to flowchart schemas adds no power, whereas the addition of a counter to $\mathcal{C}(1c)$, or to $\mathcal{C}(1\ pds)$ does indeed add power. Adding a

counter to $\mathcal{C}$(2 pds) or to $\mathcal{C}$(1q) , or to the corresponding ones with equality, adds no power because these classes seem to be omnipotent anyway as far as control capability is concerned. The features of recursion and a pushdown stack act as if they provide some control capability (to flowchart schemas), but not as much as two counters. Similarly, equality tests too provide some control capability as evidenced by the fact that a schema in $\mathcal{C}(=)$ can solve problem P_d (Example 3 in Section 2.3.2) which cannot be solved by $\mathcal{C}(R)$.

(3) The test capability. In our standard classes of schemas we placed no restriction on the kind of tests (on data items) allowed except as to whether equality tests were permitted, or were banned. Another restriction that could be placed is the maximum depth (of nesting of function symbols) of terms allowed. For example, if we allow only tests of the form $p(y)$ and $p(f(y))$ in one-variable monadic schemas without resets, we would obtain a class strictly more powerful than the Ianov schemas (which allow only $p(y)$). In general, we find that

$$\mathcal{C}(\text{n var, depth d+1}) > \mathcal{C}(\text{n var, depth d}) ,$$

and

$$\mathcal{C}(\text{n+1 var, depth d}) > \mathcal{C}(\text{n var, depth d}) .$$

These can be shown by constructing a schema quite similar to the one used in the proof of Theorem 2.14.

2.3.4 Proofs on the Power of Schemas, and Detailed Examples

2.3.4.1 Proof of Theorem 2.13

The theorem states that every schema $S_1 \in \mathcal{C}(R)$ (or in $\mathcal{C}(R, =)$) can be effectively translated into an equivalent schema S_2 in the same class

such that only data arguments are passed to the defined functions in S_2, and each defined function returns just one data value and no boolean values.

Proof:

Step 1: $S_1 \rightarrow S_3$. It is trivial to see how S_1 can be converted into an equivalent schema S_3 such that in S_3 no boolean values are passed to any defined function. This can be done as follows: if any defined function F in S_1 is passed m boolean variables, then in S_2 we have 2^m defined functions corresponding to F, one for each possible set of values the boolean variables may take. Then, if in any function definition of S_1, if F is called with some arguments, then the proper function in S_2 is called without any boolean values. This may involve testing the boolean arguments before the call (as they may be predicate or equality tests) yielding nested if-then-else's, which, of course, can then be eliminated by using additional defined functions.

Step 2: $S_3 \rightarrow S_4$. Now, given the schema S_3, we wish to convert it into an equivalent schema S_4 such that defined functions return no boolean values, only data values, and all arguments are data values too. To do this we will change the defined functions in S_3 so that they return data values instead of their boolean values. These data values will be treated as if they are really booleans by applying some fixed test on them.

We now have the problem of discovering what data values are to correspond to "true" and "false", and what fixed test we are going to use. This is the concept of finding a "locator" (Constable and Gries [1972]).

In the class $\mathcal{C}(R,=)$ this is trivial, for we can simply test to see if all zero-ary functions are equal. If they are, we apply all base functions to them to see if we can generate a new element. If not, then all terms must yield the same value, and now the outcome of the computation is quite easily determined. Otherwise, we will find two constant terms τ_1, τ_2 (of depth at most one), whose values are distinct. Then we can simply use τ_1 to stand for "true", τ_2 to stand for "false", and the test on a value x to see if x is true or not is " $x = \tau_1$ ".

In the class $\mathcal{C}(R)$, on the other hand, our problem is a little more difficult. We proceed on the lines of Constable and Gries [1972] to build a flowchart with "exits", which executes the computation of the recursive schema until it tests some predicate more than once, and it turns out to have different outcomes (true, and false) in two of the cases; in which case the flowchart exits (S_4 has one copy of S_3 for each exit). Suppose $p_i(x_1,\ldots,x_k)$ is true, and $p_i(x_1',\ldots,x_k')$ is false then the recursive schema can begin normal operation, and each defined function returns the set of vectors $x_1,\ldots,x_k$ instead of returning a true value, and returns $x_1',\ldots,x_k'$ instead of a false value. Of course, each defined function has to be passed the data values $x_1,\ldots,x_k,x_1',\ldots,x_k'$ as arguments (as well as the standard arguments). It is easy to see that a flowchart can simulate the computation of the recursive schema because if a function F_i is called recursively within another call to F_i then the arguments of the earlier call do not have to be remembered for the schema would exit before the second call "returns". Now, of course, we convert the flowchart locator into recursive definitions to get the required schema S_4.

Step 3: $S_4 \rightarrow S_2$. Finally, we translate S_4 into the desired schema S_2 where each defined function returns just one data value (and all arguments are data values too). This is done as follows. Suppose any defined function F in S_4 returns a vector of n data values, then we replace it by n functions (in S_2); call them $F_1,\ldots,F_n$. Then, any term like $Y_i(F(\ldots))$ in S_4 is replaced by $F_i(\ldots)$ in S_2 ; and of course, each F_i returns just one value -- the i-th value that F would return. That $F_i(\ldots)$ does indeed equal $Y_i(F(\ldots))$ for all arguments in the computation of the recursive schemas can be proved by induction on the depth of recursion, simultaneously for all defined functions; but we dispense with such formalism which doesn't add to the intuitive concept of the proof.

This completes the construction, and the proof. □

2.3.4.2 Proof of Theorem 2.14

To prove that there exists a schema S in $\mathcal{C}(n+1 \text{ var})$ such that no schema in $\mathcal{C}(R,n \text{ var})$ is equivalent to it.

The desired schema S is:

S: START $\langle y_0,y_1,\ldots,y_n\rangle \leftarrow \langle a,a,\ldots,a\rangle$;
$y_1 \leftarrow f(y_0)$; $y_2 \leftarrow f(y_1)$; ...; $y_n \leftarrow f(y_{n-1})$;
<u>while</u> $p(y_0) \wedge p(y_1) \wedge \ldots \wedge p(y_n)$ <u>do</u>
$\langle y_0,y_1,\ldots,y_n\rangle \leftarrow \langle g(y_0),g(y_1),\ldots,g(y_n)\rangle$;
HALT(a) .

Suppose there is a schema S_1 in $\mathcal{C}(R,n \text{ var})$ that is equivalent to S . Without loss of generality, assume that in S_1 no defined

function is passed any boolean arguments (see step 1 in the proof of Theorem 2.13). Also, without loss of generality assume that S_1 has no function other than a,f,g, and no predicate other than p (Theorem 2.1). Now, consider the computation of S_1 on a Herbrand interpretation in which all $p(x)$ are true. Then the schema S_1 must be in a "loop", that is, for some defined function F, F is called at successively larger recursion depths (possibly with different arguments) -- this is because if F calls itself recursively then the schema must loop (because F is passed no booleans, and the only tests, other than those on booleans returned, are $p(\tau)$). We define the "type" of the elements of the Herbrand domain as follows -- any element of the form $f^i(a)$, $i \leq n$, is said to be of type $(0,i)$, any element of the form $g^j f^i(a)$, $j \geq 1$, $i \leq n$, is said to be of type $(1,i)$, and all other elements have type $(0,n+1)$. Now consider two calls to F in which the types of all variables repeat. Then after the same interval they will repeat again, and again, and so on, because exactly the same sequence of "statements" are being executed. We call this a cycle of the computation. Now, as F has at most n arguments, there must be some type number $(1,m)$, $0 \leq m \leq n$, such that no argument in F has type $(1,m)$. Now, if we consider the finite interval in a cycle, only a finite number of values of type $(1,m)$ can be tested (by the predicate p) during this time, and the same values are tested over and over again. Hence, as there are only a finite number of operations executed before the cycles start, the whole computation can check only a finite number of values of type $(1,m)$. Now, if we change the interpretation slightly by making the predicate p on one of the untested values of type $(1,m)$ to be false, then the computation of S_1 must be the same as before,

that is, S_1 diverges, whereas S halts on this interpretation -- contradicting the assumption that S_1 is equivalent to S . □

2.3.4.3 Example 1 -- Inverse of a Unary Function

For simplicity we assume that the only functions are a single zero-ary function a , the given unary function f and a binary function g . The possible terms are therefore:

a , f(a) , g(a,a) , f(f(a)) , g(f(a),f(a)) , g(a,f(a)) , ...

The schema for any other set of functions is similar to the one for this particular case.

Symbols c_1, c_2, c_3 stand for counters. Strictly, the only operations allowed on counters are adding and subtracting one, and testing for zero. For convenience, however, we will also allow other statements such as $c_i \leftarrow 0$, $c_i \leftarrow c_j$, and tests like $c_i = c_j$, as it is clear that these operations can be performed using only the legal operations and additional counters.

(1) --- START $\langle y,z \rangle \leftarrow \langle a,\text{true} \rangle$, $A \leftarrow \langle a,\text{true} \rangle$;
A[0] $\leftarrow$ y;

(2) --- $c_1 \leftarrow c_2 \leftarrow 0$;

(3) --- REPEAT: $\langle y,z \rangle \leftarrow A[c_1]$;

(4) --- if f(y) = a then HALT(y);
$c_2 \leftarrow c_2+1$; $A[c_2] \leftarrow \langle f(y),z \rangle$;
$c_2 \leftarrow c_2+1$; $A[c_2] \leftarrow \langle g(y,y),z \rangle$;
$c_3 \leftarrow c_1$;

```
        while c3 ≠ 0 do
            begin
            c3 ← c3-1;
            c2 ← c2+1; A[c2] ← ⟨g(A[c3],y),z⟩
            c2 ← c2+1; A[c2] ← ⟨g(y,a[c3]),z⟩
            end;
        c1 ← c1+1;
(5) --- goto REPEAT.
```

After the initialization phase (lines (1) to (2))(ignoring all booleans):

$A[0] = a$, $c_1 = c_2 = 0$.

After completing one pass through the outer loop of the program (lines (3) to (5))

$A[1] = f(a)$, $A[2] = g(a,a)$, $c_1 = 1$, $c_2 = 2$,

and after a second pass

$A[3] = f(f(a))$, $A[4] = g(f(a),f(a))$,

$A[5] = g(a,f(a))$, $A[6] = g(f(a),a)$, $c_1 = 2$, $c_2 = 6$.

The algorithm works as follows: two pointers c_1 and c_2 reference the array. $A[c_1]$ represents the "current" value. If the current value is not an inverse, as determined by line (4), it is composed with values preceding it in the enumeration by function applications, and the new values obtained are added to the array.

It can be shown by induction that the process of enumeration generates and tests each possible term exactly once. This means that an inverse will be found if it exists. The point at which the test of the inverse is made could be changed to effect time efficiency but without altering the main features of the program.

2.3.4.4 Example 2 -- Herbrand-like Interpretations

We assume that the only functions are a single zero-ary function a, a unary function f and a binary function g. Therefore the set of terms includes

$$a\ ,\ f(a)\ ,\ g(a,a)\ ,\ f(f(a))\ ,\ g(f(a),f(a))\ ,\ g(a,f(a))\ ,\ \ldots\ ,$$

that is, the same as in the previous example. The required schema is:

(1) --- START $\langle y,y',z\rangle \leftarrow \langle a,a,\text{true}\rangle$, $A \leftarrow \langle a,\text{true}\rangle$;

$A[0] \leftarrow \langle y,z\rangle$;

(2) --- $c_1 \leftarrow c_2 \leftarrow 0$;

(3) --- REPEAT: $\langle y,z\rangle \leftarrow A[c_1]$;

(4) ---

```
c4 ← c1;
while c4 ≠ 0 do
     begin
     c4 ← c4-1; ⟨y',z⟩ ← A[c4];
     if y' = y then HALT;
     end;
```

$c_2 \leftarrow c_2+1$; $A[c_2] \leftarrow \langle f(y),z\rangle$;

$c_2 \leftarrow c_2+1$; $A[c_2] \leftarrow \langle g(y,y),z\rangle$;

$c_3 \leftarrow c_1$;

while $c_3 \neq 0$ do

begin

$c_3 \leftarrow c_3-1$;

$c_2 \leftarrow c_2+1$; $A[c_2] \leftarrow \langle g(A[c_3],y),z\rangle$;

$c_2 \leftarrow c_2+1$; $A[c_2] \leftarrow \langle g(y,A[c_3]),z\rangle$;

end;

$c_1 \leftarrow c_1+1$;

(5) --- goto REPEAT.

This program is quite similar to the previous one in the manner of enumeration of terms. The fact that each term is generated exactly once is used in making the test (4) to check if a value is repeated.

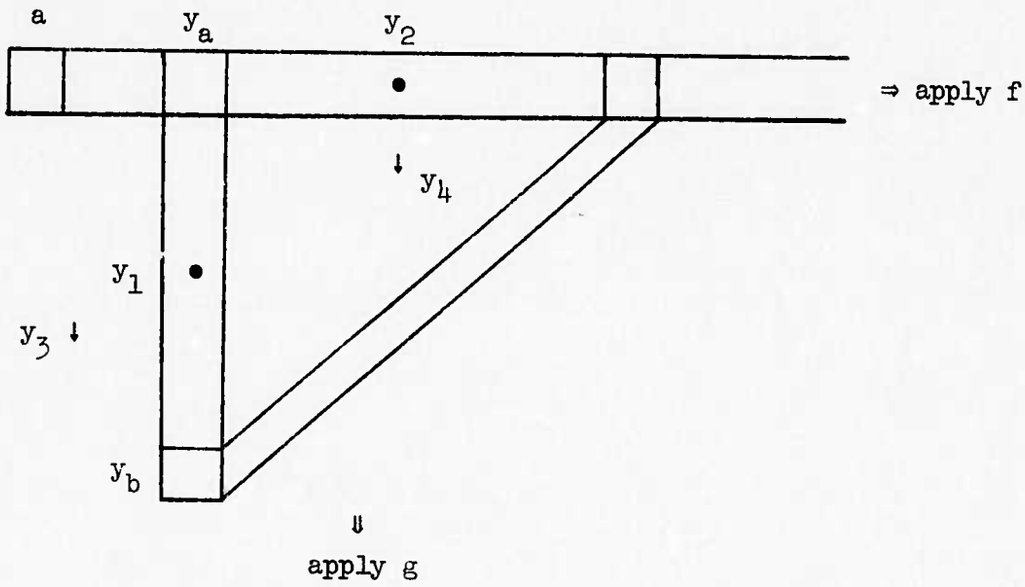
2.3.4.5 Example 3 -- The Witch Hunt

"To find an element x of the form $g^j f^i(a)$, $i,j \geq 0$, such that $p(x)$ is false". The desired schema of $\mathcal{C}(=)$ uses seven variables -- y_a, y_b, y_1, y_2, y_3, y_4, y_5.

```
(1)  --  START ⟨y_a,y_b,y_1,y_2,y_3,y_4,y_5⟩ ← ⟨a,a,a,a,a,a,a⟩;
         if ¬ p(a) then HALT(a);
(2)  --  NEXT: y_1 ← y_a;
         while y_1 ≠ y_b do
              begin if y_1 = f(y_b) then goto RESET;
                    y_1 ← g(y_1)
              end;
(3)  --  if y_b = f(y_b) then goto RESET;
         y_b ← f(y_b);
         if ¬ p(y_b) then HALT(a);
         ⟨y_1,y_2⟩ ← ⟨y_a,y_a⟩;
(4)  --  while y_1 ≠ y_b do
              begin y_1 ← g(y_1); y_2 ← f(y_2);
                    y_3 ← y_1; y_4 ← y_2;
                    while y_3 ≠ y_b do begin y_3 ← g(y_3); y_4 ← g(y_4) end;
                    if ¬ p(y_4) then HALT(a);
(5)  --       end;
         goto NEXT;
```

```
(6)  --  RESET: ⟨y1,y2⟩ ← ⟨ya,ya⟩;
         FAIL: y1 ← g(y1); y2 ← f(y2);
          y3 ← y1; y4 ← y2;
          while y3 ≠ yb do
               begin y5 ← y2;
                     while y5 ≠ y4 do
                          begin if y5 = g(y4) then goto FAIL;
                                y5 ← g(y5)
                          end;
                     if y4 = g(y4) then goto FAIL;
                     y3 ← g(y3);
                     y4 ← g(y4);
               end;
         SUCCEED: ya ← y2;
          yb ← y4;
(7)  --  goto NEXT.
```

The operation of the schema may briefly be described as follows. The schema effectively "counts" on the range of values from y_a to y_b, all of which are guaranteed to be distinct. The part of the schema between lines (2) and (3) checks to see if counting can be done on a larger domain: from y_a to $f(y_b)$. If so, then the "slice" of values shown in the figure below are tested to see if the predicate p is false for any of them.



If, however, the domain from y_a to y_b cannot be extended, then the segment of the schema from lines (6) to (7) resets y_a and y_b.

2.3.4.6 Example 4 -- Translation of Flowchart Schemas with One Counter

The recursive schema

$F_0 \Leftarrow F(a);$

$F(y) \Leftarrow$ if $p(y)$ then $f(y)$ else $F(G(f(y)));$

$G(y) \Leftarrow$ if $q(y)$ then $g(y)$ else $G(G(g(y)));$

can be translated to a flowchart schema with one program variable y and one counter c.

```
START y ← a;
while ¬ p(y) do
    begin
    y ← f(y);
    while true do
        if q(y)
            then begin
                y ← g(y);
(1) ---             if c = 0 then goto DONE;
(2) ---             c ← c-1
                end
            else begin
                y ← g(y);
(3) ---             c ← c+1
                end;
    DONE:   end;
HALT(f(y)) .
```

The corresponding equivalent flowchart schema with equality uses three variables instead of a counter:

y_a represents a zero counter,

y_c simulates the counter, and

y_t is a temporary variable.

The idea is that y_c simulates a counter by using the value $g^n(y_a)$ to represent the integer n. Therefore, the statement $y_c \leftarrow y_a$ stands for $c \leftarrow 0$, $y_c \leftarrow g(y_c)$ stands for $c \leftarrow c+1$, and the statements $[y_t \leftarrow y_a;$ while $g(y_t) \neq y_c$ do $y_t \leftarrow g(y_t);\ y_c \leftarrow y_t]$ stand for $c \leftarrow c-1$.

We have to be careful, however. The term $g^n(y_a)$ stands for the integer n, $n \geq 0$, only if for no two distinct $i,j \leq n$ are the terms $g^i(y_a)$ and $g^j(y_a)$ equal. Interpretations for which the counter is required to count up to an integer n where there exist $i,j \leq n$, $i \neq j$, such that $g^i(y_a) = g^j(y_a)$ are called looping interpretations. It is easy to see that for looping interpretations the given recursive schema never halts. The required program schema is therefore easy to construct.

START $\langle y, y_a, y_c, y_t \rangle \leftarrow \langle a, a, a, a \rangle$;
while $\neg\, p(y)$ do
    begin
    $y \leftarrow f(y)$;
    $y_a \leftarrow y$; $y_c \leftarrow y_a$;
    while true do
        if $q(y)$
            then begin
                $y \leftarrow g(y)$;
(1) ---   if $y_c = y_a$ then goto DONE;
                $y_t \leftarrow y_a$;
(2) ---   while $g(y_t) \neq y_c$ do $y_t \leftarrow g(y_t)$;
                $y_c \leftarrow y_t$
                end

```
                    else begin
                        y ← g(y);
                    ┌─────────────────────────────────────┐
                        y_t ← y_a;                        │
                        while y_t ≠ y_c do                │ check
                            if y_t = g(y_c) then LOOP     │ for a
                                                          │ looping
                                          else y_t ← g(y_t);│ inter-
                        if y_t = g(y_c) then LOOP;        │ preta-
                    └─────────────────────────────────────┘ tion
(3) ---             y_c ← g(y_c)
                    end;
            DONE:   end;
        HALT(f(y)) .
```

Note that this flowchart schema is equivalent to the given recursive schema even when the functions and predicates are not total.

Proof. $\mathcal{C}(=) \equiv \mathcal{C}(1c,=)$

Since $\mathcal{C}(=) \leq \mathcal{C}(1c,=)$, we only have to prove that $\mathcal{C}(=) \geq \mathcal{C}(1c,=)$.

Given a schema in $\mathcal{C}(1c,=)$, we reduce it to a canonical form S' (for one counter schemas) which is a recursive schema whose base functions a , f , g , and predicates p , q need not be total, and we can give a meaning to a , f , g , p , q in terms of the base functions and predicates of S that makes the schema S' equivalent to S . Further, the "meaning" for all a , f , g , p , q is flowchartable. Thus, we would find that since we have a schema S" in $\mathcal{C}(=)$ equivalent to S' , if we substitute the meanings of the functions and predicates we would obtain a schema in $\mathcal{C}(=)$ equivalent to S . For convenience below, after every statement c ← c+1 in S insert a (distinct) null statement, say

$y_1 \leftarrow y_1$. The canonical form S' below can be simplified somewhat, e.g. the term F(G(f(y))) can be changed to F(G(y)) -- we choose not to do so here. The schema S' is

S': F_0 <= F(a);

F(y) <= if p(y) then f(y) else F(G(f(y)));

G(y) <= if q(y) then g(y) else G(G(g(y))) .

We will represent the meanings of a , p , q , f , g by nonrecursive subroutines. Without loss of generality assume that there are no loop statements in S , that all halt statements are of the form HALT(y_1) , and that all statements are labeled. Suppose $\bar{y} = \langle y_1,\ldots,y_n\rangle$ and $\bar{z} = \langle z_1,\ldots,z_m\rangle$ are the variables in S . Consider any interpretation for S with domain D . Then the domain D' for S' is $D^n \times \{T,F\}^{m+\ell}$ where $\ell = \lceil \log_2 s \rceil$ and s is the number of statements (or labels) in S . We will represent an element in D' as a vector $\langle \bar{y},\bar{z},L\rangle$ where L is a label whose value corresponds to a label in S (L is to be implemented by booleans).

(i) If the start statement in S is

START $\langle \bar{y},\bar{z}\rangle \leftarrow \langle \bar{\tau}(),\bar{\alpha}()\rangle$; goto L_i

then a is $\langle \bar{\tau}(),\bar{\alpha}(),L_i\rangle$.

(ii) f($\langle \bar{y}',\bar{z}',L'\rangle$) is

begin data $\bar{y}$; boolean $\bar{z}$; label L;

$\bar{y} \leftarrow \bar{y}'$; $\bar{z} \leftarrow \bar{z}'$; $L \leftarrow L'$;

REP: goto L;

L_1: $STATEMENT_1$;

L_2: $STATEMENT_2$;

:

L_s: $STATEMENT_s$;

end .

Above, a variable declared label L represents a vector of ℓ booleans, and we allow statements like goto L where L is a label variable (it is clear how this statement is to be implemented).

	If in S we have	then STATEMENT_i is
(a)	L_i: $\langle \bar{y}, \bar{z} \rangle \leftarrow \langle \bar{\tau}, \bar{\alpha} \rangle$; goto L_j	$\langle \bar{y}, \bar{z} \rangle \leftarrow \langle \bar{\tau}, \bar{\alpha} \rangle$; $L \leftarrow L_j$; goto REP
(b)	L_i: if α then goto L_j else goto L_k	if α then $L \leftarrow L_j$ else $L \leftarrow L_k$; goto REP
(c)	L_i: HALT(y_1)	RETURN($\langle \bar{y}, \bar{z}, L \rangle$)
(d)	L_i: $c \leftarrow c+1$; goto L_j	RETURN($\langle \bar{y}, \bar{z}, L_j \rangle$)
(e)	L_i: $c \leftarrow c-1$; goto L_j	$L \leftarrow L_j$; goto REP
(f)	L_i: if $c = 0$ then goto L_j else goto L_k	$L \leftarrow L_j$; goto REP

(iii) $g(\langle \bar{y}', \bar{z}', \bar{L}' \rangle)$ is like the function f except for parts (e), (f):

(e)	L_i: $c \leftarrow c-1$; goto L_j	RETURN($\langle \bar{y}, \bar{z}, L_j \rangle$)
(f)	L_i: if $c = 0$ then goto L_j else goto L_k	$L \leftarrow L_k$; goto REP .

(iv) $p(\langle \bar{y}', \bar{z}', L' \rangle)$ is

```
begin data ȳ; boolean z̄; label L;
    ⟨ȳ, z̄, L⟩ ← f(⟨ȳ', z̄', L'⟩);
    if isplus(L) then RETURN(false)
                 else RETURN(true)
end .
```

Above, the function isplus(L) is defined to be true if L is the label L_j in a statement $c \leftarrow c+1$; goto L_j, and false otherwise.

(v) $q(\langle \bar{y}', \bar{z}', L' \rangle)$ is

```
begin data ȳ; boolean z̄; label L;
    ⟨ȳ, z̄, L⟩ ← g(⟨ȳ', z̄', L'⟩);
    if isplus(L) then RETURN(false)
                 else RETURN(true)
end .
```

If the value computed by F(a) is $\langle y_1, \ldots, y_n, z_1, \ldots, z_m, L \rangle$ then y_1 represents the output of S.

S' implements the computation of the one-counter schema S by representing the value of the counter by the number of defined functions G that effectively exist in the recursion stack at any time. When the defined function F is being "executed" the counter is zero.

This shows how to convert any schema in $\mathcal{C}(1c,=)$ to an equivalent schema in $\mathcal{C}(=)$, which completes the proof.

2.3.4.7 Proof of Theorem 2.16

$\mathcal{C}(R,=) \not\geq \mathcal{C}(c)$

We will use the fact that schemas in $\mathcal{C}(c)$ can simulate Turing machines, and that the halting of schemas in $\mathcal{C}(R,=)$ over a given finite domain is decidable, to demonstrate a diagonalized argument.

The required schema $S \in \mathcal{C}(3c)$ is defined as follows.

The schema S uses three counters. The initialization phase of S is the following:

START $y \leftarrow a$;

<u>while</u> $p(y)$ <u>do begin</u> $y \leftarrow f(y)$; $c \leftarrow c+1$ <u>end</u>;

After this phase the schema makes no use of the variable y, or the base functions or predicate (except in the halt statement). Let the value of the counter c be n when it exits from the initialization phase. Let I_n denote the following interpretation: the domain has n+1 elements,

$$e_0, e_1, \ldots, e_n ,$$

the value of a is e_0, and f and p are defined by:

$$i < n \quad f(e_i) = e_{i+1} \qquad p(e_i) = \text{true}$$

$$f(e_n) = e_0 \qquad p(e_n) = \text{false} .$$

The schema S then simulates the computation of the n-th schema S_n in $\mathcal{C}(R, =)$ on the interpretation I_n. The schema S_n will diverge if and only if some defined function calls itself recursively with exactly the same arguments (data and boolean values). If S_n halts with output a, then S loops, otherwise S halts with output a.

This completes the description of the required schema S; and it is clear that it is not equivalent to any schema in $\mathcal{C}(R, =)$, because if it were equivalent to, say, the m-th schema, we find their outputs on the interpretation I_m disagree -- a contradiction.

$\mathcal{C}(c,=) \not\geq \mathcal{C}(R)$

We can show the equivalent result that there is a schema in $\mathcal{C}(R)$ not equivalent to any in $\mathcal{C}(c,=)$. It is

S: $F_0 <= F(a)$;

$F(y) <=$ *if* $p(y)$ *then* $h(F(f(y)),F(g(y)))$ *else* y .

This is the schema demonstrated by Paterson and Hewitt [1970], and their proof, shown for $\mathcal{C}()$, also applies to $\mathcal{C}(c,=)$.

Let τ_0 be defined to be the term a , and τ_{n+1} to be the term $h(f(\tau_n),g(\tau_n))$. Also define the Herbrand interpretation H_n to be: $p(\tau)$ is false if the depth of nesting of function symbols in τ is n , otherwise it is true. Then, $Val(S,H_n) = \tau_n$. Now, suppose there is a schema $S' \in \mathcal{C}(c,=)$ equivalent to S . Without loss of generality, we can restrict all terms appearing in S' to have depth at most one (depth of terms a, y_i is 0 , of terms $f(a)$, $f(y_i)$ is 1 , of $f(f(a))$, $h(f(a),y)$ is 2 , etc.). Then we see that if S' has m data variables then S' cannot compute any terms τ_n if $n > m$ (for Herbrand interpretations). Thus the outputs of S and S' over H_{m+1} must disagree and S and S' cannot be equivalent.

2.3.4.8 Proof of Theorem 2.19

To prove that

(a) $\mathcal{C}(R) \equiv \mathcal{C}(1\ \text{pds}) \equiv \mathcal{C}(1\ \text{list})$, and

(b) $\mathcal{C}(R, =) \equiv \mathcal{C}(1\ \text{pds}, =) \equiv \mathcal{C}(1\ \text{list}, =)$.

$\mathcal{C}(R) \leq \mathcal{C}(1\ \text{pds})$, $\mathcal{C}(R, =) \leq \mathcal{C}(1\ \text{pds}, =)$

We do not describe the construction in detail because it is obvious. Given a recursive schema S we construct a schema with a stack S' as follows: S' can stack boolean variables to code any finite piece of information. S' has a set of variables that represent the arguments of a function call, another set to represent the values returned, and some for temporaries. When a recursive call is to be made, the old arguments and some temporaries (values of earlier calls from the same defined function -- required to build up terms) are stacked, as well as the local context, the new arguments are set up, and computation is begun on the new defined function. When a function returns, values (and the context) are unstacked. S' halts if the stack becomes empty.

$\mathcal{C}(1\ \text{pds}) \leq \mathcal{C}(R)$, $\mathcal{C}(1\ \text{pds}, =) \leq \mathcal{C}(R, =)$

Given a schema S with one pushdown stack, we construct a recursive schema S' equivalent to S , such that S' uses equality tests only if S uses them. For the sake of convenience, we will allow certain features in recursive schemas that are not strictly allowed, but can be easily eliminated to get a legal recursive schema. These include the following:

(1) Nested if-then-else's .

(2) Passing labels as parameters (arguments and values returned) and nonlooping goto-statements in a recursive definition. Labels can be implemented by a vector of booleans, and transfers can be implemented by nested if-then-else's . We also allow return-statements which explicitly return values from the defined functions.

Without loss of generality, S has a single halt statement of the form HALT(y_1) , and has no loop statements (L_i: LOOP can be replaced by L_i: $\langle \bar{y}, \bar{z} \rangle \leftarrow \langle \bar{y}, \bar{z} \rangle$; goto L_i). In the schema S we label all assignment statements, test statements, the halt statement, and also all statements operating on the stack as follows:

```
s ← push(s,y,z)

if s = Λ then goto L
     else begin
          ⟨y,z⟩ ← top(s);
          s ← pop(s)
          end
```

```
L_i: s ← push(s,y,z)

L_i: if s = Λ
L_j: then goto L
          else begin
               ⟨y,z⟩ ← top(s);
               s ← pop(s);
               end
```

Notice the strange placement of the label L_j after the test $s = \Lambda$. In addition, we have a dummy label L_{halt} which is assumed to be entered after the halt statement.

The recursive schema S' has four defined functions:

F_0 -- The starting function. This calls F_Λ .

F_Λ -- When this is executed the stack is empty. F_Λ may call itself iteratively (i.e., a compiler can treat it as iteration

rather than recursion). It returns only when the schema S halts. F_Λ may also call the function F .

F -- This is the work-horse. This is similar to the function used in converting a flowchart schema into a recursive schema. It calls F_s when something is pushed into the stack.

F_s -- The number of recursive calls on F_s represents the height of the pushdown stack.

These functions are defined as follows. Recall that the notation $Y_i(\ldots)$ is used to pick the i-th data element of a vector. We also use $\bar{Y}(\ldots)$ to pick all data elements from a vector, and $\bar{Z}(\ldots)$ to pick all boolean elements. Similarly, we will use the notation $L(\ldots)$ to pick the label from a vector (only one will be used).

$\underline{F_0}$: If the start statement in S is

START $\langle \bar{y}, \bar{z} \rangle \leftarrow \langle \bar{\tau}, \bar{\alpha} \rangle$; goto L_i;

then

$$F_0 \Leftarrow Y_1(F_\Lambda(\bar{\tau}, \bar{\alpha}, L_i));$$

$\underline{F_\Lambda}$: $F_\Lambda(\bar{y}, \bar{z}, L) \Leftarrow$ goto $L(F(\bar{y}, \bar{z}, L))$;

L_1: return(exp_1);

L_2: return(exp_2);

⋮

For any i :

(1) If L_i is the dummy statement L_{halt} then the expression exp_i is $F(\bar{y}, \bar{z}, L)$.

(2) If L_i is the "weird" label in

L_i: **if** s = Λ
then **goto** L_j
else **begin** ⟨y,z⟩ ← top(s); s ← pop(s) **end**

then the expression exp_i is

$$F_\Lambda(\bar{Y}F(\bar{y},\bar{z},L),\bar{Z}F(\bar{y},\bar{z},L),L_j) \quad .$$

The value returned by F will have the same type, i.e., $\langle \bar{y},\bar{z},L\rangle$, and it represents the "current" values of the variables and the label of the next statement to be executed. Notice that the effect of exp_i is to stick the values returned back into F (in the next call to F_Λ) and continue the execution from where it was left off.

(3) If L_i is anything else -- this can never happen, and exp_i is arbitrary.

F : F(y,z,L) <= **goto** L;
L_1: **return**(exp_1);
L_2: **return**(exp_2);
⋮

If L_i is	then exp_i is
(1) The dummy statement L_{halt} or the halt statement itself	$\langle \bar{y},\bar{z},L_{halt}\rangle$
(2) L_i: $\langle \bar{y},\bar{z}\rangle \leftarrow \langle \bar{\tau},\bar{\alpha}\rangle$; **goto** L_j;	$F(\bar{\tau},\bar{\alpha},L_j)$
(3) L_i: **if** α **then** **goto** L_j **else** **goto** L_k	**if** α **then** $F(\bar{y},\bar{z},L_j)$ **else** $F(\bar{y},\bar{z},L_k)$

rather than recursion). It returns only when the schema S halts. F_Λ may also call the function F .

F -- This is the work-horse. This is similar to the function used in converting a flowchart schema into a recursive schema. It calls F_s when something is pushed into the stack.

F_s -- The number of recursive calls on F_s represents the height of the pushdown stack.

These functions are defined as follows. Recall that the notation $Y_i(\ldots)$ is used to pick the i-th data element of a vector. We also use $\bar{Y}(\ldots)$ to pick all data elements from a vector, and $\bar{Z}(\ldots)$ to pick all boolean elements. Similarly, we will use the notation $L(\ldots)$ to pick the label from a vector (only one will be used).

$\underline{F_0}$: If the start statement in S is

START $\langle \bar{y}, \bar{z} \rangle \leftarrow \langle \bar{\tau}, \bar{\alpha} \rangle$; goto L_i;

then

$$F_0 \Leftarrow Y_1(F_\Lambda(\bar{\tau}, \bar{\alpha}, L_i));$$

$\underline{F_\Lambda}$:

$$F_\Lambda(\bar{y}, \bar{z}, L) \Leftarrow \underline{\text{goto}}\ L(F(\bar{y}, \bar{z}, L));$$

$$L_1: \underline{\text{return}}(\text{exp}_1);$$

$$L_2: \underline{\text{return}}(\text{exp}_2);$$

$$\vdots$$

For any i :

(1) If L_i is the dummy statement L_{halt} then the expression exp_i is $F(\bar{y}, \bar{z}, L)$.

(2) If L_i is the "weird" label in

L_i: **if** s = Λ
then **goto** L_j
else **begin** ⟨y,z⟩ ← top(s); s ← pop(s) **end**

then the expression exp_i is

$$F_\Lambda(\bar{Y}F(\bar{y},\bar{z},L),\bar{Z}F(\bar{y},\bar{z},L),L_j) \quad .$$

The value returned by F will have the same type, i.e., $\langle\bar{y},\bar{z},L\rangle$, and it represents the "current" values of the variables and the label of the next statement to be executed. Notice that the effect of exp_i is to stick the values returned back into F (in the next call to F_Λ) and continue the execution from where it was left off.

(3) If L_i is anything else -- this can never happen, and exp_i is arbitrary.

F:

```
F(y,z,L) <= goto L;
        L1: return(exp1);
        L2: return(exp2);
            ⋮
```

If L_i is	then exp_i is
(1) The dummy statement L_{halt} or the halt statement itself	$\langle\bar{y},\bar{z},L_{halt}\rangle$
(2) L_i: $\langle\bar{y},\bar{z}\rangle \leftarrow \langle\bar{\tau},\bar{\alpha}\rangle$; **goto** L_j;	$F(\bar{\tau},\bar{\alpha},L_j)$
(3) L_i: **if** α **then goto** L_j **else goto** L_k	**if** α **then** $F(\bar{y},\bar{z},L_j)$ **else** $F(\bar{y},\bar{z},L_k)$

(4) L_i: s ← push(s,y,z); goto L_j $\qquad$ $F_s(\bar{y},\bar{z},L_j,y,z)$

(5) L_i: if s = Λ $\qquad$ $(\bar{y},\bar{z},L_j)$

L_j: then goto ...

The only case not shown cannot occur, and the expression for it is arbitrary.

F_s: $\qquad F_s(\bar{y},\bar{z},L,y,z) \Leftarrow$ goto $L(F(\bar{y},\bar{z},L))$;

L_1: return(exp_1);

L_2: return(exp_2);

⋮

(Note -- there should be no ambiguity as to the roles of the two L's above). For any i, if L_i is a statement of the form

if s = Λ

L_i: then goto L_j

else begin $\langle y_k, z_m \rangle$ ← top(s); s ← pop(s) end

then the corresponding exp_i is

$F(\bar{y}',\bar{z}',L_j)$

where $\bar{y}'$ is obtained by substituting y for y_k in the vector $F(\bar{y},\bar{z},L)$; and $\bar{z}'$ is obtained by substituting z for z_m in the same vector. If this has caused any confusion, it may be pointed out that y_k really stands for the k-th data element, and similarly for z_m.

The only other possible case is that L_i is L_{halt}, and in this case exp_i is $F(\bar{y},\bar{z},L)$.

□

$\mathcal{C}$(1 pds) $\leq \mathcal{C}$(1 list) , $\mathcal{C}$(1 pds), =) $\leq \mathcal{C}$(1 list, =)

A pushdown stack can be simulated by a list as follows. In the construction below, L' is an arbitrary label (transfers to L' can never be taken in actual computation), and y' is a dummy variable. The list schema uses a zero-ary function a to represent "true", and Λ to represent "false".

Pushdown stack	List
s ← push(s,y,z)	ℓ ← cons(y,ℓ); if z then ℓ ← cons(a,ℓ) else ℓ ← cons(Λ,ℓ)
if s = Λ then goto L	if ℓ = Λ then goto L;
else	if atom(ℓ) then goto L'
begin	else if ¬ atom(car(ℓ)) ∨ car(ℓ) = Λ
⟨y,z⟩ ← top(s);	then goto L_2
s ← pop(s)	else y' ← car(ℓ);
end	z ← true;
	goto L_3;
	L_2: z ← false;
	if atom(ℓ) then goto L'
	else ℓ ← cdr(ℓ);
	L_3: if atom(ℓ) then goto L'
	else if ¬ atom(car(ℓ)) ∨ car(ℓ) = Λ
	then goto L'
	else y ← car(ℓ);
	if atom(ℓ) then goto L'
	else ℓ ← cdr(ℓ)

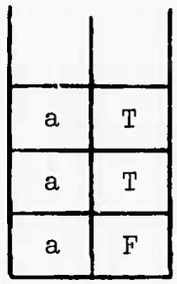
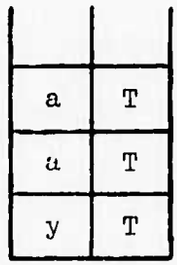
□

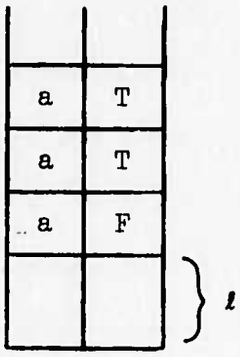
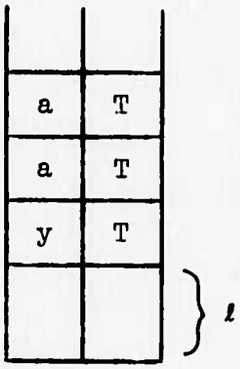
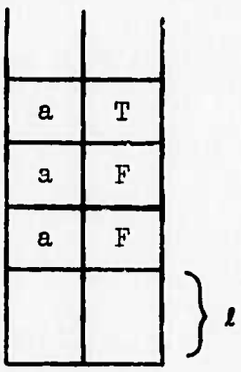
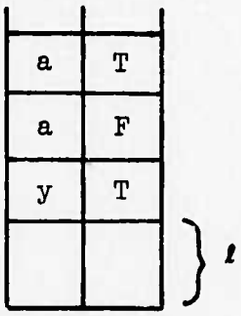
$\mathcal{C}(1\ \text{list}) \leq \mathcal{C}(1\ \text{pds})$, $\mathcal{C}(1\ \text{list}, =) \leq \mathcal{C}(1\ \text{pds}, =)$

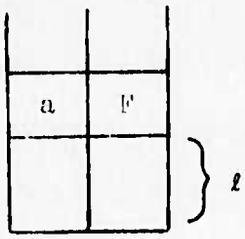
The list operations can be simulated by a stack as follows. The top three pairs in the stack represent either the car or the cdr of the list, the rest of the stack represents the rest of the list. The only exception to this is when both the car and the cdr of the list are lists. When a schema has just one list l , this can only happen by the execution of a statement

$$l \leftarrow \text{cons}(l, l)$$

that is, the car and the cdr are the same. This is represented by a boolean value in the stack that represents a "doubled" list. The representation of a list by a stack can be done as follows (a is any zero-ary function):

list	stack representation
Λ	
an atom - y	

list	stack representation
$\Lambda.\ell$	
$y.\ell$	
$\ell.\Lambda$	
$\ell.y$	

list	stack representation
$\ell . \ell$	

Note: the stack representation of a list is not unique, but depends on the way the list is built up. Now, it is clear how the list can be manipulated by its stack representation. We have been able to represent the list by a stack because a schema with a single list cannot generate lists of any great complexity.

□

2.3.4.9 Proof of Theorem 2.20 (Maximal Classes of Schemas)

$\mathcal{C}(1\ \text{pds}, \text{lc}) \geq \mathcal{C}(\text{pds}, \text{q}, \text{list}, \text{A})$, $\mathcal{C}(1\ \text{pds}, \text{lc}, =) \geq \mathcal{C}(\text{pds}, \text{q}, \text{list}, \text{A}, =)$

We first demonstrate that a schema with a pushdown stack and two counters can simulate the computation of any schema S with any number of features -- pushdown stacks, queues, lists, arrays, counters. We will take recourse to the large body of knowledge on the programming of Turing machines (Church's thesis).

Now, two counters can simulate a Turing machine computation (on a blank tape). We are using the term "Turing machine" somewhat loosely here because we will allow the machine to output as it computes, and also in some special state to accept a yes-no input (from the environment) before deciding what to do next. Our two-counter Turing machine will keep track of the values in all the pushdown stacks, queues, lists, arrays, and counters of the schema S. Data values will be kept in symbolic form, that is, as (constant) terms. Of course, an infinite amount of memory is not required to keep track of arrays -- the Turing machine need only remember those array locations that were assigned to since the beginning of the computation, and know about the value the array was initialized to by the start statement. If S executes a test on data elements (a predicate or equality test), then the Turing machine "outputs" a list of instructions as to how all terms are to be constructed and the test to be made -- the output is a postfix-polish form of the expression (it uses only constant terms -- no variables). Postfix polish can be executed on the pushdown stack and the outcome of the test is transmitted to the Turing machine. Our two-counter machine can output one character (say, the n-th character) as follows: if c_1, c_2 are the counters, c_1 is set to $2^n.k$ where k is some odd integer, and c_2 is 0 (see the construction of a two-counter machine from a multi-counter machine in the discussion on Theorem 2.17). The output can then be detected by:

```
if (c1 mod 2) = 1 then goto OUTPUT0;
c1 ← c1/2;
if (c1 mod 2) = 1 then goto OUTPUT1;
```

$c_1 \leftarrow c_1/2$;

if $(c_1 \bmod 2) = 1$ then goto OUTPUT2;

⋮

where it is obvious how the test $(c_1 \bmod 2) = 1$, and the assignment $c_1 \leftarrow c_1/2$ can be implemented.

Now, the schema in $\mathcal{C}(1\ \text{pds}, 2c)$ we obtain has the following interesting property. Whenever it executes a statement like

$s \leftarrow \text{push}(s, y, z)$

or like

if $s = \Lambda$ then goto L

else begin $\langle y, z \rangle \leftarrow \text{top}(s)$; $s \leftarrow \text{pop}(s)$ end

the value of the counter c_2 is zero. Hence we can implement c_2 in the stack itself by stacking a false value to represent $c_2 = 0$, and subsequently a true value for each increment to the value of c_2. This will not interfere with the above stack operations since we simply throw away the false value, execute the stack operation, and then reinstate it.

We note that if the functions of the schema are monadic, then $\mathcal{C}(2c)$ can simulate $\mathcal{C}(\text{pds}, q, \text{list}, A)$ (and similarly for $\mathcal{C}(2c, =)$). In the above description of a schema with one stack and two counters, the stack was only used to construct (constant) terms. When the functions are monadic, any term can be computed with just one variable, and hence any n-ary predicate test can be performed with n-variables. This shows that $\mathcal{C}(2c, \text{monadic fns}) \equiv \mathcal{C}(\text{pds}, q, \text{list}, A, \text{monadic fns})$, and that $\mathcal{C}(2c, =, \text{monadic fns}) \equiv \mathcal{C}(\text{pds}, q, \text{list}, A, =, \text{monadic fns})$.

□

$\mathcal{C}(\text{lq}) \geq \mathcal{C}(\text{1 pds, lc})$, $\mathcal{C}(\text{lq}, =) \geq \mathcal{C}(\text{1 pds, lc}, =)$

Since a pds is at least as powerful as a counter, it suffices to show that $\mathcal{C}(\text{lq}) \geq \mathcal{C}(\text{2 pds})$, $\mathcal{C}(\text{lq}, =) \geq \mathcal{C}(\text{2 pds}, =)$ (the proof is a little simpler). Given a schema S with two stacks s_1 and s_2 , we wish to construct a schema S' with a queue that is equivalent to S . But this is easy because both stacks can be packed in a queue, with boolean variables to mark the ends, and the values can be circulated. The detailed construction is as follows. For convenience below, we use the notation tf(1) for "true", and tf(2) for "false", and we define macros rem(L,y,z,q) , and reset(i) as follows:

```
rem(L,y,z,q)        if q = Λ then goto L else
                       begin ⟨y,z⟩ ← first(q);
                             q ← remove(q)
                       end

reset(i)          L:rem(L',y',z',q);
                    q ← add(q,y',z');
                       begin rem(L',y',z',q);
                             q ← add(q,y',z');
                             goto L
                       end;
                    rem(L',y',z',q);
                    add(q,y',z');
                    if z' ≠ tf(i) then goto L
```

where L' is an arbitrary label, " a " is a zero-ary function in S , and y' and z' are new variables in the schema S' (with the queue) that are not present in S .

Schema S -- two stacks (s_1, s_2)

START $\langle y_1, y_2, \ldots, z_1, z_2, \ldots \rangle$
$\leftarrow \langle \tau_1, \tau_2, \ldots, \alpha_1, \alpha_2, \ldots \rangle$

$s_i \leftarrow \text{push}(s_i, y, z)$

if s = Λ then goto L else
begin ⟨y,z⟩ ← top(s);
s ← pop(s);
end

Schema S' -- one queue

START $\langle y', y_1, y_2, \ldots, z', z_1, z_2, \ldots \rangle$
$\leftarrow \langle a, \tau_1, \tau_2, \ldots, \text{true}, \alpha_1, \alpha_2, \ldots \rangle$;
q ← add(q,a,false);
q ← add(q,a,tf(1));
q ← add(q,a,false);
q ← add(q,q,tf(2))

reset(i);
q ← add(q,a,true);
q ← add(q,y,z)

reset(i);
rem(L',y',z',q);
if ¬ z' then
begin q ← add(q,y',z');
rem(L',y',z',q);
q ← add(q,y',z');
goto L
end;
rem(L',y,z,q)

□

Chapter 3 Decision Problems

3.1 Introduction

We consider the following decision problems for classes of schemas:

(a) The halting problem -- to decide whether a given schema in the class halts on every interpretation.

(b) The divergence problem -- to decide whether a given schema in the class diverges on every interpretation.

(c) The equivalence problem -- to decide whether two schemas are equivalent (decide if $S_1 \equiv S_2$).

(d) The inclusion problem -- given two schemas S_1 and S_2 to decide whether it is true that for every interpretation either both schemas halt with the same output or S_2 diverges (decide if $S_1 \geq S_2$).

(e) The isomorphism problem -- to decide whether two schemas are isomorphic to each other (decide if $S_1 \sim S_2$).

It should be stated that for "conventional" schemas, i.e., all schemas introduced in the previous chapter, the problems (a)-(e) are in general unsolvable, but the following problems are partially solvable:

(a) The halting problem -- to decide whether a given schema in the class halts on every interpretation.

(b') The nondivergence problem -- to decide whether a given schema ever halts.

(e') The nonisomorphism problem -- to decide if two schemas are not isomorphic to each other.

The notable exceptions are the equivalence and inclusion problems. In general, the equivalence and inclusion problems as well as their

negations are all not partially solvable.

A class of schemas is said to be solvable if its decision problems (a)-(e) are solvable; similarly, a class is unsolvable if its decision problems (a)-(e) are unsolvable. Of course, some classes may be neither solvable, nor unsolvable.

The class of Ianov schemas, which consists of one-variable flowchart schemas using only monadic functions and predicates and no resets is solvable. However, even very simple classes of two-variable schemas are unsolvable. For example, the class of schemas with one constant a, one other function symbol f, one predicate p, and statements of the forms:

(1) START $\langle y_1, y_2 \rangle \leftarrow \langle a, a \rangle$

(2) HALT(a)

(3) LOOP

(4) $y_i \leftarrow f(y_i)$

(5) if $p(y_i)$ then goto L_1 else goto L_2

is unsolvable. For this reason, in this chapter we will almost exclusively consider schemas with only one variable to determine how large a class can be constructed before it becomes unsolvable.

Also note that for solvability considerations the use of boolean variables is irrelevant as they can be eliminated. Hence we will only consider schemas without boolean variables.

In Section 3.2 we consider uninterpreted one-variable flowchart schemas in which equality tests are allowed. In view of the fact that all decision problems for uninterpreted one-variable schemas without equality tests are solvable, it may be somewhat unexpected that the class of one-variable schemas with general equality tests is unsolvable. But we show that if only some restricted equality tests are allowed the resulting classes are solvable.

In Section 3.3 we consider some semi-interpreted schemas, in particular, those obtained when (a) two unary functions are specified to commute, and (b) when some unary function is invertible, i.e., composition of the function with its inverse is the identity function. We find that with commutativity or invertibility alone, the decision problems are solvable, but if both are allowed, they become unsolvable.

3.2 Equality Tests

3.2.1 Notation

We consider flowchart schemas with a single variable y, and we use the symbols

(1) $a, a_1, a_2, \ldots$ to represent individual constants (or zero-ary functions, if you will),

(2) $f, f_1, f_2, \ldots$ to represent n-ary functions $(n \geq 1)$, and

(3) $p, p_1, p_2, \ldots$ to represent n-ary predicates $(n \geq 0)$.

We use the notation $\tau()$ to represent a constant term, i.e., a term not containing the variable y, and τ, $\tau(y)$ to represent an arbitrary term.

The assignment depth $\|\tau(y)\|$ of a term $\tau(y)$ is defined as follows:

$\|\tau()\| = 0$,

$\|y\| = 0$,

$\|f_i(\tau_1, \ldots, \tau_r)\| = \max\{\|\tau_1\|, \ldots, \|\tau_r\|\} + 1$, where at least one of the τ_j's is nonconstant.

The depth $|\tau|$ of a term τ is the maximum depth of nesting in the term, and is defined by:

$$|a_i| = 0 ,$$

$$|y| = 0 ,$$

$$|f_i(\tau_1,\ldots,\tau_r)| = \max\{|\tau_1|,\ldots,|\tau_r|\}+1 .$$

We also say that $|\tau|$ is the depth of nesting of τ .

Note that for nonconstant monadic terms τ , $\|\tau\| = |\tau|$, but in general $\|\tau\| \leq |\tau|$. For example, $\|f(g(a),y)\| = 1$, but $|f(g(a),y)| = 2$.

3.2.2 Solvable Classes

Consider the rather general class $\mathcal{C}_1$ of flowchart schemas with one variable. Schemas in $\mathcal{C}_1$ contain the following statement types (L_1 and L_2 are arbitrary labels in the definitions below):

Start statement:	START $y \leftarrow a_i$
Final statements:	HALT(τ) or LOOP
Assignment statement:	$y \leftarrow \tau$
Predicate-test statement:	if $p_i(\tau_1,\ldots,\tau_n)$ then goto L_1 else goto L_2
Equality-test statement:	if $\tau_1 = \tau_2$ then goto L_1 else goto L_2

The equality tests allowed must, however, satisfy the condition that either τ_1 or τ_2 is a constant term, or else there exist terms $\tau_1'(y)$, $\tau_2'(y)$ such that both $\|\tau_1'(y)\|$ and $\|\tau_2'(y)\|$ are less than or

equal to 1 , and τ_1 and τ_2 are of the forms $\tau_1'(\tau)$, and $\tau_2'(\tau)$ respectively for some term τ . Note: $\tau_1'(\tau)$ is a term obtained from $\tau_1'(y)$ by substituting all occurrences of y simultaneously by τ ; and similarly for $\tau_2'(\tau)$. Note also, that as a special case of this condition, tests of the form $\tau_1 = \tau_2$ with both $\|\tau_1\|, \|\tau_2\| \leq 1$ are allowed (simply by choosing τ to be the term y itself). Another example of a test that is allowed by this condition: $f(\tau) = \tau$, where f is some unary function and τ is an arbitrary term -- this is allowed because we can choose τ_1' to be $f(y)$ and τ_2' to be y .

Theorem 3.1 (Solvability of $\mathcal{C}_1$)

The class $\mathcal{C}_1$ is solvable, i.e., for $\mathcal{C}_1$:

(a) the halting problem is solvable;

(b) the divergence problem is solvable;

(c) the equivalence problem is solvable;

(d) the inclusion problem is solvable;

(e) the isomorphism problem is solvable.

This theorem includes as special cases the results of Ianov [1960], Rutledge [1964], and also recent extensions by Pnueli [private communication], and Garland and Luckham [1971]. The proof is presented in Section 3.2.4.

As a special case of this theorem, the class of all one-variable schemas without equality tests, $\mathcal{C}$(1 var) , is solvable.

As another special case, the class of one-variable monadic schemas allowing resets, and equality tests of the forms:

$$\tau_1() = \tau_2 \quad , \quad y = f_i(y) \quad , \quad \text{and} \quad f_i(y) = f_j(y)$$

is solvable.

Consider, next, the class $\mathcal{C}_2$ of schemas, similar to the class $\mathcal{C}_1$, but with a change in the form of equality tests allowed, viz., the equality test statements allowed are of the form:

$$\underline{\text{if}}\ \tau_1 = \tau_2\ \underline{\text{then goto}}\ L_1\ \underline{\text{else goto}}\ L_2\ ,$$

but this time the restriction is that either τ_1 or τ_2 is a constant term, or else $\|\tau_1\| = \|\tau_2\|$.

Theorem 3.2 (Solvability of $\mathcal{C}_2$)

The class $\mathcal{C}_2$ is solvable.

As a special case, the class of one-variable monadic schemas allowing resets and equality tests of the forms:

$$\tau_1(y) = \tau_2()\ ,\ \text{or}\ \tau_1 = \tau_2\ \text{where}\ |\tau_1| = |\tau_2|$$

is solvable.

3.2.3 Unsolvable Classes

It should well be asked why we have the "strange" restrictions on the form of equality tests above. The answer is that even slight generalizations of the restrictions above yield, astonishingly, classes whose problems are unsolvable. We demonstrate this on two classes.

Consider the class $\mathcal{C}_3$ consisting of one variable y, one constant a, no predicates and only monadic function constants. Statements in schemas of $\mathcal{C}_3$ are of the forms:

Start statement:	START $y \leftarrow a$
Final statements:	HALT(a) or
	LOOP

Assignment statement: $y \leftarrow f_i(y)$

Equality-test statement: $\underline{\text{if}}\ f_i(y) = f_j(f_k(y))\ \underline{\text{then goto}}\ L_1$ $\underline{\text{else goto}}\ L_2$

$\mathcal{C}_3$ differs from $\mathcal{C}_1$ in that terms of assignment depth two are effectively used in equality tests; and it differs from $\mathcal{C}_2$ in that terms tested for equality do not have the same assignment depth.

Theorem 3.3 (Unsolvability of $\mathcal{C}_3$)

The class $\mathcal{C}_3$ is unsolvable, i.e., for $\mathcal{C}_3$:

(a) the halting problem is unsolvable;

(b) the divergence problem is not partially solvable;

(c) the equivalence problem is not partially solvable;

(d) the inclusion problem is not partially solvable;

(e) the isomorphism problem is not partially solvable.

For the sake of completeness we should mention that the non-equivalence and the non-inclusion problems for this class too are not partially solvable. Of course, the halting, non-divergence and non-isomorphism problems are partially solvable, which follows from the general result mentioned in Section 3.1. For the proof, see Section 3.2.4.

We introduce, next, the class $\mathcal{C}_4$ of one-variable monadic schemas similar to $\mathcal{C}_3$ but with the difference that equality tests allowed have the following form:

$\underline{\text{if}}\ y = \tau\ \underline{\text{then goto}}\ L_1\ \underline{\text{else goto}}\ L_2$

where τ may have any of the forms:

$f_i(y)$,

$f_i(f_j(y))$,

or

$f_i(f_j(f_k(y)))$.

Theorem 3.4 (Unsolvability of $\mathcal{C}_4$)

The class $\mathcal{C}_4$ is unsolvable.

Classes $\mathcal{C}_1$ and $\mathcal{C}_2$ are solvable, whereas $\mathcal{C}_3$ and $\mathcal{C}_4$ are unsolvable. On comparing these classes it is clear that there is a very sharp demarcation between classes of one-variable schemas that are solvable, and those that are unsolvable, depending on the form of equality tests allowed. It should perhaps be asked how many function symbols suffice to render a class unsolvable. It can be shown, for example, that for the class $\mathcal{C}_3$, merely four functions are sufficient. It is more interesting to note, however, that these function symbols can be "coded" using only two function symbols so that schemas with one variable, two functions and general equality tests, i.e., tests of the form $\tau_1(y) = \tau_2(y)$, are unsolvable. Note: the number of functions does not include the ever-present constant (or zero-ary function) a .

So far we have restricted our consideration to schemas that have only one variable. The reason is obvious: one-variable schemas provide the most interesting solvable classes. When more variables are allowed, even a very few features tend to make the schemas unsolvable. For example, schemas with two variables, two functions and tests only of the form $y_i = f(y_i)$ are unsolvable.

It is even more interesting, though probably not surprising, that schemas with a single function too are unsolvable; for example, the class of one function schemas having tests only of the form $y_i = y_j$ is unsolvable (four variables suffice in this case).

The proofs of these secondary results are also presented in Section 3.2.4.

3.2.4 Proofs for Schemas with Equality

3.2.4.1 Proof of Theorem 3.1 (Solvability of $\mathcal{C}_1$)

For convenience, in this proof we change our notation for terms very slightly: τ stands for an arbitrary term and $\tau()$ stands for a constant term as before, but $\tau(y)$ represents a non-constant term.

3.1(a), (b), (c) The solvability of the halting, divergence and equivalence problems follows from the solvability of inclusion:

(a) Given a schema S of $\mathcal{C}_1$, S halts if and only if $S' \geq H$ where H represents the schema START; HALT(a) that always halts with output a, and S' is the schema S with all halt statements changed to HALT(a).

(b) Given a schema S of $\mathcal{C}_1$, S diverges if and only if $L \geq S$, where L represents the schema START; LOOP that always loops.

(c) Given two schemas S_1 and S_2 of $\mathcal{C}_1$, $S_1 \equiv S_2$ if and only if $S_1 \geq S_2$ and $S_2 \geq S_1$.

3.1(d) To show the solvability of the inclusion problem we will first present a proof for schemas in $\mathcal{C}_1$ using only monadic functions

and predicates, and then indicate how it may be extended to include non-monadic functions and predicates as well.

We first describe classes of canonical interpretations that play a role for the monadic schemas in $\mathcal{C}_1$ similar to the role of Herbrand interpretations for Herbrand schemas (see Theorem 2.1.2).

For any integer $k \geq 0$, we describe the class of interpretations $\mathcal{I}_k$ (over a set of monadic functions and predicates) as follows. The elements of the domain of an interpretation $I \in \mathcal{I}_k$ are equivalence classes of constant terms. However, each constant term need not be present in some equivalence class. First, consider the set of terms $\tau()$ such that $|\tau()| \leq k$. Equivalence classes may consist of arbitrary non-overlapping subsets of these terms as long as substitutivity relations are preserved, for example, if $k \geq 3$, and $f(g(a_1))$, $f(a_2)$ are in the same equivalence class, then $f(f(g(a_1)))$, $f(f(a_2))$ must together be in some class, as must $g(f(g(a_1))$, $g(f(a_2))$, but $g(a_1)$, a_2 may be in different classes. All constant terms $\tau()$, with $|\tau()| \leq k$ are in some equivalence class, and these are called the <u>initial elements</u> of $Dom(I)$. We will rank the terms in an equivalence class first by depth, and then by (some) lexicographic order, and choose the smallest as the representative of the class. We denote a class by $[\tau()]$ where $\tau()$ is the representative. Also, if $\tau()$ is any element in a class, not necessarily its representative, we use $\{\tau()\}$ to denote the class. Since the equivalence classes will be non-overlapping, these notations make sense.

Functions are defined on the initial elements in the obvious way. If $|\tau()| < k$ then $f([\tau()]) = \{f(\tau())\}$. If all initial elements are

of the form $[\tau()]$ with $|\tau()| < k$, then there are no other elements in Dom(I). Otherwise, if $[\tau()]$ is an element of Dom(I), $|\tau()| \geq k$, then new equivalence classes may consist of terms from the set $\{f(\tau()) \mid f \text{ is a unary function symbol}\}$, and for any function symbol f, if there is a class, of which $f(\tau())$ is an element, then $f([\tau()]) = [f(\tau())]$, otherwise $f([\tau()])$ is either $[\tau()]$, or some initial element.

All predicates on Dom(I) are arbitrary.

This defines the class of interpretations $\mathcal{I}_k$.

Now, given an arbitrary interpretation I', we define the corresponding interpretation I in $\mathcal{I}_k$ (notation $I' \overset{k}{\rightarrow} I$) in the obvious way. Two terms are in the same equivalence class (in I) only if their corresponding values are equal (but the converse is not necessary). We have, in addition, the following rules:

(1) for any $\tau_1()$, $\tau_2()$, such that $|\tau_1()| \leq k$, $|\tau_2()| \leq k$, the two terms are in the same equivalence class in I if and only if their values are equal in I'.

(2) If $[\tau()]$, $[\tau'()]$ are classes in I such that $|\tau()| \geq k$, $|\tau'()| \leq k$, then if the values of $f(\tau())$ and $\tau'()$ are equal in I' then $f([\tau()]) = [\tau'()]$ in I.

(3) If $[\tau()]$ is a class in I, and $\tau()$ and $f(\tau())$ are equal in I' then $f([\tau()]) = [\tau()]$ in I.

(4) If $[\tau()]$ is a class in I such that in I', the value of $f(\tau())$ equals the value of $g(\tau())$, and $f(\tau())$ does not equal $\tau'()$, for any $\tau'()$ with $|\tau'()| \leq k$, then in I the terms $f(\tau())$ and $g(\tau())$ are in the same equivalence class.

(5) If $[\tau()]$ is a class in I, then $p([\tau()])$ is true in I if and only if $p(\tau())$ is true in I'.

By the construction of interpretations in $\mathcal{I}_k$, this describes a unique I corresponding to I', and a homomorphism Θ from I onto the reachable elements (i.e., elements that can be represented as constant terms) of I'.

Lemma. Given any monadic schema $S \in \mathcal{C}_1$, and an integer k such that for every term τ used in S, $|\tau| \leq k$, then for any interpretation I' for S, if $I' \xrightarrow{k} I$ and Θ is the homomorphism $\Theta : I \to I'$, then

(1) $\text{Path}(S,I') = \text{Path}(S,I)$, and

(2) $\text{Val}(S,I') = \Theta(\text{Val}(S,I))$ if both are defined.

Proof: The lemma follows by induction on the number of steps in the simultaneous computation of S on I' and on I with the induction hypothesis that after n steps, the paths are the same and the values of the variable y in the two computations are related by Θ.

It follows from this lemma that to prove halting, divergence, equivalence, isomorphism or freedom, it suffices to prove these for the interpretations $\mathcal{I}_k$ (for appropriate k) because if the outputs of two schemas on an interpretation I' are distinct, they are also distinct on the corresponding interpretation I.

This result (for inclusion and isomorphism) is used throughout in the proof below, where whenever we say "an interpretation", we mean an interpretation from the class $\mathcal{I}_k$.

Given two monadic schemas, change all assignment statements $y \leftarrow \tau(y)$ so that the only kinds of assignment statements are of the form $y \leftarrow f_i(y)$ or $y \leftarrow a_i$, and halts are of the form HALT(y). Let the resulting schemas be called S_1, S_2. To explain the algorithm for deciding whether or not $S_1 \geq S_2$, we first introduce the concept of a state vector.

Given an interpretation for the schemas S_1, S_2 and a value for the variable y, we define the specification state of the variable y to mean the true/false values of all predicate and equality tests the schema(s) could possibly make without changing the value of the variable y. To make this notion concrete, let k be the maximum depth of any term used in the schemas S_1 and S_2. Given a value $[\tau()]$ for y, the specification state of y includes the following:

(1) the description of all initial elements and all equivalence classes of the form $[\tau'(\tau())]$ where $|\tau'(y)| \leq k$;

(2) the values of all terms $\tau'(y)$ where $|\tau'(y)| \leq k$; and

(3) the values of all atomic formulas $p(\tau'(y))$ for all p, and $|\tau'(y)| \leq k$.

We define the incomplete specification state like the specification state except that k is replaced by $k-1$ in the definition above. We define the state vector of the variable y to be the incomplete specification state as well as the current statement just executed.

Now, given the two schemas S_1 and S_2 we construct a finite state automaton which effectively simulates the computations of S_1 and S_2 in parallel. The input tape represents an interpretation (from $\mathfrak{I}_k$) for the schemas S_1, S_2, appropriately coded. The automaton accepts

the input tape unless either (i) both schemas halt with different outputs, or (ii) S_2 halts and S_1 either loops or can be made to diverge. The finite automaton can detect the latter case (for the appropriate input tape) because the "principal instance" of the second schema will enter the same state vector twice after the first schema has halted. Now, the finite state automaton accepts all input tapes if and only if $S_1 \geq S_2$.

The description of the automaton and the input tape follows. The automaton effectively simulates the computations of the schemas by running the computations for a (large) number of instances of the variable y in parallel. For each assignment statement in the schemas and each constant term $\tau()$, where $|\tau()| \leq k$ there is an instance of y which indicates the computation as would be executed starting just after that statement and with the variable y set to value of $\tau()$. In addition, there is a principal instance for each schema. It corresponds to the start statement and the initial value of y, i.e., it corresponds to the "real" computation of the schema. As the automaton steps through the two schemas (as determined by its input tape) the automaton keeps track of a finite amount of bookkeeping information, viz., the various instances that have equal values, the various instances that halt or loop forever, and, of course, the state vectors for instances that have not halted or looped up to that point (called active instances). In addition, the automaton remembers the initializing character (explained below), and if S_2 has halted, then it also keeps track of the set of state vectors of the principal instance of S_1 subsequent to the halting of S_2.

The first character of the input tape is a special charachter called the initializing character. It describes all elements of the form $[\tau()]$, where $|\tau()| \leq 2k\text{-}1$, and gives the values of all terms $\tau()$, and all atomic formulas like $p(\tau())$, where $|\tau()| \leq 2k\text{-}1$. With this amount of information the automaton can simulate the execution of all instances of y so that for each instance either it halts or loops or reaches a value $[\tau()]$ such that $|\tau()| = k$.

All subsequent characters on the input tape are called updating characters. If m is the number of instances in S_1 and S_2, and we let X denote the finite set of specification states, then an updating character is an element of X^m. In other words, one updating character provides the following information for each instance in both schemas:

(1) the description of all "new" equivalence classes, i.e., for all classes $[\tau(y)]$, $|\tau(y)| = k\text{-}1$, and all function symbols f, the description of equivalence classes amongst the terms of the form $f_i(\tau(y))$;

(2) the values of all terms $\tau(y)$, $|\tau(y)| = k$; and

(3) the values of all atomic formulas $p_i(\tau(y))$, $|\tau(y)| = k$.

When an updating character is read, the automaton already has an incomplete specification state for each instance. If for any active instance, the information given by the updating character fails to match the incomplete specification state for that instance (and the information of the initializing character), the automaton detects the tape as representing an infeasible interpretation. Whenever any infeasible interpretation is detected, the input tape is accepted. Further, the automaton checks that the "updates" are equal for instances known to have equal values --

otherwise the interpretation is infeasible. If the updating character passes these "feasibility" tests the automaton then steps each active instance through the schema in which that instance occurs. The following cases are possible:

(1) The next statement is a HALT or a LOOP statement -- record it. The instance becomes inactive, but all instances that become inactive by halting with this value are remembered in the finite memory.

(2) The next statement is a test statement -- the outcome is known, hence continue the process (check for a loop).

(3) The next statement is $y \leftarrow a_i$ -- the instance becomes identical with the instance that started from this statement with value a_i (check for a loop).

(4) The next statement is $y \leftarrow f_i(y)$ --

(a) If $y = f_i(y)$ then y is unchanged -- continue the process, checking for a loop.

(b) $y \neq f_i(y)$, $f_i(y) = \tau()$ with $|\tau()| \leq k$ -- the instance becomes identical with the instance that started from this statement with value $\tau()$.

(c) $y \neq f_i(y)$, $f_i(y) \neq \tau()$ for all $\tau()$ such that $|\tau()| \leq k$ -- the process stops.

The automaton continues reading input characters until either both S_1 and S_2 halt or loop, or until S_2 loops (while S_1 is still active). If, however, S_2 halts and S_1 is still active, all state vectors for the principal instance of S_1 are remembered and if it ever loops or repeats a state the input tape is rejected.

The reason that this specification state approach works with limited equality tests is that the finite specification state carries sufficient information to allow it to be updated such that all feasible updates represent feasible interpretations. The converse, that for every feasible interpretation there is a feasible update at each step, is trivial. This is not true for general equality tests, e.g., in the classes $\mathcal{C}_3$ and $\mathcal{C}_4$ if a specification state were to carry all information necessary to update it, the amount of information would grow without bound as the computation proceeded.

To generalize to non-monadic schemas in $\mathcal{C}_1$, we describe the canonical interpretations $\mathcal{I}_k$ similar to those for monadic schemas.

The elements of the domain are, as before, equivalence classes over terms. There is, however, a special element denoted by $[\Lambda]$. This corresponds to terms that cannot be built up. For any interpretation in $\mathcal{I}_k$, the value of all functions having $[\Lambda]$ as any argument is $[\Lambda]$; and the value of all predicates having $[\Lambda]$ as any argument is (arbitrarily) true. We now describe the other elements in Dom(I). The "initial elements" are the equivalence classes over all terms $\tau()$ where $|\tau()| \leq k$, satisfying substitutivity, of course. As before, we rank terms first by $|\tau()|$, and then by (some) lexicographic order, and we use the notations $[\tau()]$ and $\{\tau()\}$ as before.

Functions over initial elements are defined as follows. If all $|\tau_1()|, \ldots, |\tau_r()| < k$, then $f([\tau_1()], \ldots, [\tau_r()]) = \{f(\tau_1(), \ldots, \tau_r())\}$, where f is an r-ary function. If $[\tau()]$ is in Dom(I), $|\tau()| \geq k$, then new equivalence classes may consist of terms from the set T of terms $\tau'(\tau())$ where $\tau'(y)$ is a non-constant term with $|\tau'(y)| \leq k$, as follows:

(1) Let $T_1 \subset T$ be the set of terms $\tau'(\tau())$ where $\|\tau'(y)\| = 1$, and where $\tau'(y)$ is (non-constant and) of the form $f(\tau_1,\ldots,\tau_r)$ where for each i, either τ_i is simply y or else τ_i is a constant term and $[\tau_i]$ is an initial element. Then equivalence classes on T_1 are arbitrary, and we define the value of $f([\tau_1],\ldots,[\tau_r])$ to be [*] $\{f(\tau_1,\ldots,\tau_r)\}$ if such a class exists, otherwise it is either $[\tau()]$ or some initial element.

(2) Let $T_2 \subset T$ be the set of terms $\tau'(\tau())$ where $\|\tau'(y)\| = 2$, $\tau'(y)$ is of the form $f(\tau_1,\ldots,\tau_r)$ where for each i, $[\tau_i]$ is an equivalence class (at least for some i, $\|\tau_i\| = 1$) and there exist non-constant τ_i, τ_j for which $\tau_i \neq \tau_j$. Then for each term $\tau'(\tau()) \in T_2$ there is a class $[\tau'(\tau())]$ consisting of just the singleton, and the value of $f([\tau_1],\ldots,[\tau_r])$ is defined to be this element.

(3) $T_3,\ldots,T_k$ may generate additional new elements in a manner similar to (2) above.

All function applications not specified above have value $[\Lambda]$, and all predicates taking arguments from $\mathrm{Dom}(I)-[\Lambda]$ are arbitrary.

This defines the class of interpretations $\mathfrak{I}_k$, and for monadic functions and predicates it is the same as the earlier class $\mathfrak{I}_k$ introduced (except for the unreachable element $[\Lambda]$).

Now, given an arbitrary interpretation I', we obtain the corresponding $I \in \mathfrak{I}_k$ $(I' \overset{k}{\rightarrow} I)$ as before, having the property that there is a surjection $\Theta\colon \mathrm{Dom}(I)-[\Lambda] \rightarrow D$ that preserves the values of predicates

---

[*] With a little corrupted notation we have allowed $[y]$ to stand for $[\tau()]$ where $\tau()$ is the value of y, and we continue to use y and $\tau()$ interchangeably.

and functions. Here, D is the set of k-reachable elements in Dom(I') which is defined to be the set of elements in Dom(I') corresponding to the terms $\tau()$ that can be built up by assignments: $y \leftarrow \tau_1(); y \leftarrow \tau_2(y); \ldots; y \leftarrow \tau_n(y)$, where for all i, $|\tau_i| \leq k$. The desired lemma can then be proved, that is, if every term τ used in Sch_1 has depth at most k, then if $I' \overset{k}{\rightarrow} I$ then $Path(S,I') = Path(S,I)$, and $Val(S,I') = \Theta(Val(S,I))$. The rest of the proof is almost identical to the proof above, except that we cannot impose that all instances can be simulated exactly in step, but some instances may get up to a bounded number (k-1) of steps ahead of others -- but this is no problem, the automaton simply remembers these relationships, and always updates those (active) instances lagging behind.

This completes the proof of inclusion. But before the reader starts sharpening his pencil to write a program for proving the equivalence of programs by this method, a note of caution seems to be in order. The size of the automaton grows quite rapidly with the size of the input schemas. Perhaps the verb "explode" would be more appropriate. To decide if $S_1 \geq S_2$ where both S_1, S_2 are the trivial schema

```
START y ← a; HALT(y)
```

the automaton is trivial. But if we add an assignment statement and change the schema to

```
START y ← a;
while p(y) do y ← f(y);
HALT(y)
```

then the automaton (in a brute force construction) has some 30 billion states and an alphabet of size 500 million. Of course, large improvements

are possible to make the decision procedure feasibly in practice by more careful definitions of canonical interpretations, specification states, and the automaton construction (e.g. if the automaton merely counts the number of steps of S_1 after S_2 halts, instead of keeping track of all state vectors entered), but that is not our purpose in the proof.

3.1(e) The proof of isomorphism is similar to the proof of inclusion, except that the automaton not only keeps track of which instances are equal in value at each step, but also which equal instances have an isomorphic history. The automaton can then detect if for any input tape the computations of the two schemas are not isomorphic.

3.2.4.2 Proof of Theorem 3.2 (Solvability of $\mathcal{C}_2$)

The proof of Theorem 3.2 is somewhat similar to that for Theorem 3.1, but the canonical interpretations and the automaton to be constructed have to be a little more general. Intuitively, the reason for this is the following. For schemas in the class $\mathcal{C}_1$, if two instances "diverge" in their values, then from that point onwards their predicate and equality tests are independent of each other. Not so for schemas in $\mathcal{C}_2$. For a schema in $\mathcal{C}_2$, two instances may diverge and then come together again, for example, the following may happen. We denote two instances by y_1 and y_2; then say, both are equal, and one, say y_1, tests $f(f(f(f(y_1)))) = f(f(f(g(y_1))))$, and it is true. Then y_1 applies $y_1 \leftarrow f(y_1)$ and y_2 applies $y_2 \leftarrow g(y_2)$, namely, they diverge. But they can converge again if the function f is applied three times to each.

We will demonstrate a quick proof for the inclusion problem. The solvability of halting, divergence and equivalence follow from this, and isomorphism can be shown to be solvable in much the same way.

Given (monadic or non-monadic) schemas $S_1, S_2 \in \mathcal{C}_2$, to decide if $S_1 \geq S_2$ we describe the canonical interpretations for S_1, S_2. Let k be the maximum depth of any term used in S_1 or in S_2. We define the effective assignment depth $//\tau()//$ of constant terms $\tau()$ as follows:

$$//\tau()// = \underline{\text{if}}\ |\tau()| \leq k\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ |\tau()|-k\ .$$

The canonical interpretations $\mathcal{I}_k$ are defined as follows. The domain of an interpretation $I \in \mathcal{I}_k$ is equivalence classes over all constant terms, but all elements of an equivalence class must have the same effective assignment depth, and equivalence classes must satisfy substitutivity. The values of functions are defined in the obvious way, that is, $f([\tau_1],\ldots,[\tau_r])$ is $\{f(\tau_1,\ldots,\tau_r)\}$ if such a class exists, otherwise it is some initial element; and the predicates are arbitrary. It is to be noted that all equivalence classes are finite, but unbounded, i.e., the input tape of the automaton to be constructed cannot specify the entire description of the elements, but that will not be necessary.

The automaton simulates the computation of all instances in parallel keeping a total specification state instead of specification states for each instance. Let $Y = \{y_1,\ldots,y_m\}$ denote the set of all instances. The total specification state contains the following:

(1) a map $D: Y \to \{0,1,\ldots,k-1\}^m$ giving the relative effective assignment depths of all instances (at least one of which is zero),

(2) the values of all: $\tau_1(y_i) = \tau_2(y_j)$, where $\tau_1(y)$, $\tau_2(y)$ are non-constant terms, and $\|\tau_1(y)\| + D(y_i) = \|\tau_2(y)\| + D(y_j) \leq k$, i.e., the effective assignment depths of both $\tau_1(y_i)$ and $\tau_2(y_j)$ are the same (because we will have that the values of y_i , y_j have depth $\geq k$),

(3) the values of all: $\tau_1(y_i) = \tau_2()$, where $\|\tau_1(y)\| + D(y_i) \leq k$, $|\tau_2()| \leq k$, and

(4) the values of all $p(\tau_1,\ldots,\tau_r)$ where $\tau_1,\ldots,\tau_r$ are all terms on some y_i , (or constant), and for non-constant τ_j , $|\tau_j| + D(y_i) \leq k$, and for constant τ_j , $|\tau_j| \leq k$.

The rest of the execution of the automaton, i.e., the initialization, updating, simulation and halting, is on the lines of the earlier proof.

3.2.4.3 Proof of Theorem 3.3 (Unsolvability of $\mathcal{C}_3$)

3.3(a),(b) We define a class $\mathcal{C}_5$ of schemas having two variables y_1 and y_2 , and whose statements consist of the following:

Start statement: START $\langle y_1,y_2\rangle \leftarrow \langle a,a\rangle$

Final statements: HALT(a) or
LOOP

Test statement: $y \leftarrow f(y_i)$;
if $p(y_i)$ then goto L_j else goto L_k;

It was shown by Luckham, Park and Paterson [1970] that the halting problem for the class $\mathcal{C}_5$ is unsolvable, and that the divergence problem is not partially solvable.

To show the halting problem for $\mathcal{C}_3$ to be unsolvable we reduce the halting problem for $\mathcal{C}_5$ to that for $\mathcal{C}_3$; that is, we describe an algorithm that takes any schema S_5 in the class $\mathcal{C}_5$ as input and yields a schema S_3' in the class $\mathcal{C}_3$ such that S_3' halts if and only if S_5 halts. Similarly, to show that the divergence problem for $\mathcal{C}_3$ is not partially solvable we describe an algorithm that takes S_5 as input and yields as output a schema S_3'' in the class $\mathcal{C}_3$ such that S_3'' diverges if and only if S_5 diverges. We will unify the construction for the two cases by constructing for both cases a schema S_3 in the class $\mathcal{C}_3$ but augmented with a special final statement called the reject statement:

REJECT statement: REJECT .

The reject statement signifies that the interpretation is unacceptable and is rejected. The idea is the following. There exists a map from interpretations of S_3 that are not rejected onto the interpretations of S_5 such that the computation for S_3 under an interpretation halts if and only if the computation for S_5 under the corresponding interpretation halts.

Now it is clear that if we replace all reject statements in S_3 by HALT statements to get S_3' , then S_3' halts on every interpretation if and only if S_5 halts on every interpretation. Similarly, if we replace all reject statements by loop statements to get S_3'' , then S_3'' diverges on every interpretation if and only if S_5 diverges on every interpretation.

Given a schema S_5 in $\mathcal{C}_5$ we construct the corresponding schema S_3 in $\mathcal{C}_3$ (with the addition of REJECT statements) as follows. We use

the variable y of S_3 to represent the latest variable tested in S_5, i.e., y_1 or y_2. The function f plays the same role in S_3 as in S_5. We use a new function g called a "test function") and tests of the form

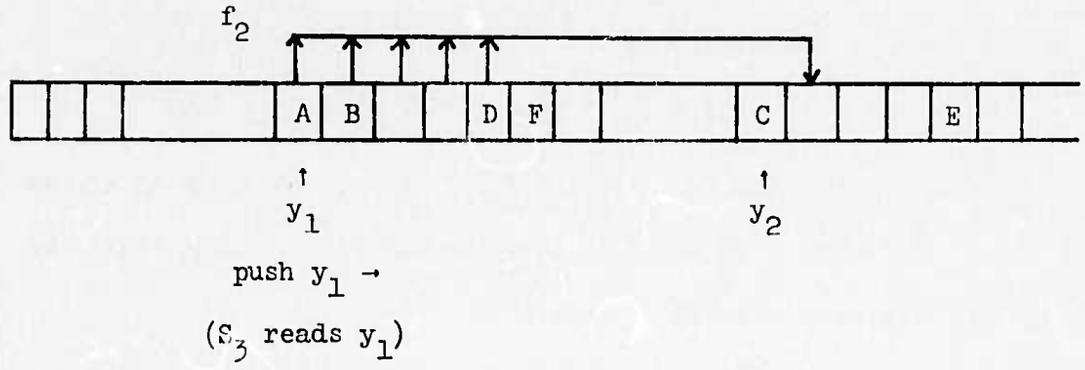
if $p(y)$ then ... else ...

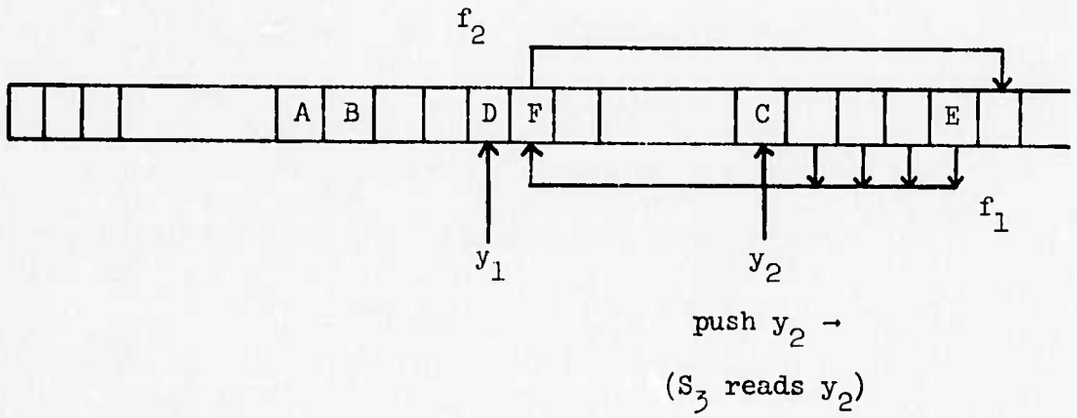
in S_5, will take the form

if $g(y) = g(g(y))$ then ... else ...

in S_3. In addition we use two "control" functions f_1 and f_2. Their roles are the following: if y stands for y_2 (of S_5) then $f_1(y)$ will equal the value of $f(y_1)$ at that instant in the computation unless, of course, a reject statement is reached earlier. The role of f_2 is analogous, i.e., if y stands for y_1 then $f_2(y)$ will equal the value of $f(y_2)$.

The schema S_3 simulates a computation of S_5 as follows. In the diagram below the elements a, $f(a)$, $f(f(a))$, $f(f(f(a)))$ are represented by contiguous squares from left to right. We superimpose on this diagram the computations of both S_3 and S_5. Suppose, at some instant in the computation of S_5, y_1 is at point A, and y_2 is at C, and suppose y_1 is being "read". S_3 makes certain that the f_2 pointers from the squares scanned, point to the right of y_2. Suppose that when y_1 reaches point D the schema S_5 starts "reading" from y_2. S_3 checks that the f_1 pointers from the squares scanned, point to the right of D (i.e., to F).





We are now in a position to describe the construction of S_3. Without loss of generality we will assume that in S_5 the first test statement tests the variable y_1. S_3 will effectively contain two copies of S_5, except, of course, for the start statement. We will call these copies A and B. We will label statements of S_5 by $L_1, L_2, L_3, \ldots$. The corresponding statements in S_3 will be labelled $AL_1, BL_1, AL_2, BL_2, AL_3, BL_3, \ldots$.

(i) The start statement in S_5 is

START $\langle y_1, y_2 \rangle \leftarrow \langle a, a \rangle$;

<u>goto</u> L_i;

The corresponding statements in S_3 are:

START $y \leftarrow a$;

if $f(y) \neq f_2(y)$ then REJECT else goto AL_i;

Note that the test $f(y) \neq f_2(y)$ is not strictly an allowed statement. We use this form for clarity: it can really be "simulated" by the statements:

if $f(y) \neq f_1(f_1(y))$ then REJECT;

if $f_2(y) \neq f_1(f_1(y))$ then REJECT else goto AL_i;

(ii) For any tests statement L_i in S_5, if L_i is of the form:

L_i: $y_1 \leftarrow f(y_1)$;

if $p(y_1)$ then goto L_j else goto L_k;

the corresponding statements AL_i and BL_i are:

AL_i: if $f_2(y) \neq f_2(f(y))$ then REJECT;

$y \leftarrow f(y)$;

if $g(y) = g(g(y))$ then goto AL_j else goto AL_k;

and

BL_i: if $f(y) \neq f_2(f_1(y))$ then REJECT;

$y \leftarrow f_1(y)$;

if $g(y) = g(g(y))$ then goto AL_j else goto AL_k;

(iii) For any tests statement L_i in S_5 of the form:

L_i: $y_2 \leftarrow f(y_2)$;

if $p(y_2)$ then goto L_j else goto L_k;

AL_i and BL_i are similar to the above, except, one has to interchange f_1 with f_2 and A with B .

(iv) Halt and loop statements remain unchanged.

This completes the construction.

The main reason that the schema S_3 can simulate the computation of S_5 is that each f_1, f_2 "pointer" is checked at most once from each square. If pointers were to be checked twice and it turned out that they were required to point to different values there might exist no interpretation satisfying this condition -- the result would be that all interpretations of S_3 would be rejected.

3.3(c) The non-partial solvability of the equivalence problem follows directly from the non-partial solvability of the divergence problem (part (b)), since a program schema in $\mathcal{C}_3$ diverges if and only if it is equivalent to the schema:

START $y \leftarrow a$;

LOOP .

3.3(d) The non-partial solvability of the inclusion problem follows immediately from the non-partial solvability of the equivalence problem since $S_1 \equiv S_2$ if and only if $S_1 \geq S_2$ and $S_2 \geq S_1$.

3.3(e) The non-partial solvability of the isomorphism problem also follows directly from the non-partial solvability of the divergence problem. Given a schema S in the class $\mathcal{C}_3$, construct a new schema S' also in $\mathcal{C}_3$ obtained by replacing each halt statement in S by the statements:

$y \leftarrow f(y)$;

HALT(a) .

Then S and S' are isomorphic if and only if S diverges.

3.2.4.4 Proof of Theorem 3.4 (Unsolvability to $\mathcal{C}_4$)

The proof goes along lines quite similar to the proof for Theorem 3.3. We first define a subset $\mathcal{C}_6$ of the class of schemas $\mathcal{C}_5$. Schemas in $\mathcal{C}_6$, like those in $\mathcal{C}_5$, have two variables y_1 and y_2, one function symbol f, and one predicate symbol p. However, $\mathcal{C}_6$ has the constraint that in any path through a schema of $\mathcal{C}_6$, after each statement that tests the variable y_1 there must be either one or two statements that test y_2 (followed by a halt or loop statement or another test of y_1) -- note the form of the test statement of $\mathcal{C}_5$ defined in the proof of Theorem 3.3(a), (b). Each "statement" in $\mathcal{C}_6$ (other than a start, halt, or loop) is a compound statement of any of the following two forms (labels $L, L_1, L_2, \ldots$ are arbitrary):

```
L: y1 ← f(y1);
   if p(y1) then
      begin
         y2 ← f(y2); if p(y2) then goto L1 else goto L2
      end
   else
      begin
         y2 ← f(y2); if p(y2) then goto L3 else goto L4
      end;
```

and

```
L: y1 ← f(y1);
   if p(y1) then
      begin
         y2 ← f(y2);
         if p(y2) then
            begin
               y2 ← f(y2); if p(y2) then goto L1 else goto L2
            end
         else
            begin
               y2 ← f(y2); if p(y2) then goto L3 else goto L4
            end
      end
   else
      begin
         ... copy of the above, except exits are L5-L8
      end;
```

Lemma. The class $\mathcal{C}_6$ is unsolvable.

Proof: The proof of unsolvability of $\mathcal{C}_6$ is similar to the proof of the unsolvability of the class $\mathcal{C}_5$. The class $\mathcal{C}_5$ is analogous to the class of two-headed automata. On the other hand, the class $\mathcal{C}_6$ corresponds to a restricted class of two-headed automata in that after each time head #1 reads a character from a binary alphabet, head #2 reads one or two characters; then head #1 reads again. Thus it is clear that head #1

can get at most one character ahead of head #2. This restricted two-headed automaton can simulate a Turing machine computation for an appropriately coded input tape as follows. The input represents a sequence of "instantaneous-descriptions" of the Turing machine computation, but between any two consecutive instantaneous descriptions are a sequence of incomplete descriptions, each one bit longer than the previous. Now, on lines similar to Luckham, Park and Paterson [1970] the restricted two-headed automaton accepts an input tape if and only if it represents the Turing machine computation alluded to above. The unsolvability of $\mathcal{C}_6$ is now obvious.

Now, given a schema S_6 in $\mathcal{C}_6$ we construct a schema S_4 in $\mathcal{C}_4$ (with reject statements) as follows. This time S_4 will have just one "copy" of S_6, but will have six function symbols: f, g, f_1, f_2, f_3, f_4.

(i) The start statement in S_6 is

START $\langle y_1, y_2 \rangle \leftarrow \langle a, a \rangle$;

goto L;

The corresponding statement in S_4 is:

START $y \leftarrow a$;

if $y \neq f_1(y)$ then REJECT;

goto L;

(ii) The statement in S_4 corresponding to a test statement of the first kind is:

L: if $y \neq f_2(f(f_1(y)))$ then REJECT;

$y \leftarrow f(f_1(y))$; comment: short for $y \leftarrow f_1(y)$ and $y \leftarrow f(y)$;

if $y = g(y)$ then

```
            begin
               if y ≠ f1(f(f2(y))) then REJECT;
               y ← f(f2(y));
               if y = g(y) then goto L1 else goto L2
            end
         else
            begin
               if y ≠ f1(f(f2(y))) then REJECT;
               y ← f(f2(y));
               if y = g(y) then goto L3 else goto L4
            end;
```

(iii) The statement in S_4 corresponding to a test statement of the second kind is:

```
L: if y ≠ f2(f(f1(y))) then REJECT;
   y ← f(f1(y));
   if y = g(y) then
      begin
         if y ≠ f3(f(f2(y))) then REJECT;
         y ← f(f2(y));
         if y = g(y) then
            begin
               if y = f4(f3(y)) then REJECT;
               y ← f3(y);
               if y ≠ f1(f(f4(y))) then REJECT;
               y ← f(f4(y));
               if y = g(y) then goto L1 else goto L2
            end
```

```
else
   begin
      if y ≠ f4(f3(y)) then REJECT;
      y ← f3(y);
      if y ≠ f1(f(f4(y))) then REJECT;
      y ← f(f4(y));
      if y = g(y) then goto L3 else goto L4
   end
end
else
   begin
      ... as above, but with exits L5-L8
   end;
```

This proves the unsolvability of 3.4(a), (b), and the parts (c), (d), and (e) are immediate from these. □

3.2.4.5 Proofs of Secondary Results

In the following results the number of functions does not include the ever present zero-ary function.

(i) Schemas with one variable, two functions and general equality tests.

The class of flowchart schemas with one variable, two functions (no predicates) and general equality tests is unsolvable.

If completely general equality tests are allowed it is easy to see that two function constants suffice to render the class of schemas unsolvable because more function letters can be "coded" in terms of two

functions. In the proof of Theorem 3.3 we change the construction of S_3 from S_5, somewhat, by making the following substitutions: for all terms τ, simultaneously substitute

$f(f(\tau))$ for $f(\tau)$

$f(g(\tau))$ for $g(\tau)$

$g(f(\tau))$ for $f_1(\tau)$

$g(g(\tau))$ for $f_2(\tau)$

All the unsolvability results go through on making this substitution.

(ii) Schemas with two variables, two functions and restricted equality tests.

The class of flowchart schemas with two variables and two functions (no predicates) with tests only of the form $y_i = f(y_i)$ are unsolvable.

Consider the class $\mathcal{C}_7$ which is the same as $\mathcal{C}_5$ but with the difference that there are two functions f and g, and no predicate constant.

Every schema S_5 in $\mathcal{C}_5$ can be reduced to an equivalent schema S_7 in $\mathcal{C}_7$ by replacing every test statement of the form

$y_i \leftarrow f(y_i)$;

if $p(y_i)$ then goto L_j else goto L_k

by a test statement of the form

$y_i \leftarrow f(y_i)$;

if $y_i = g(y_i)$ then goto L_j else goto L_k .

It is easy to see that for any finite or infinite path through S_5, if there exists an interpretation for which S_5 executes statements along this path, then there is an interpretation for which S_7 executes

statements along the corresponding path. This establishes the unsolvability of the class c_7 .

(iii) Schemas with one function, restricted equality tests.

Schemas with one function using tests only of the form $y_i = y_j$ are unsolvable.

Consider the class of two-counter programs having statements of the following kinds:

(1) START $\langle c_1, c_2 \rangle \leftarrow \langle 0,0 \rangle$

(2) $c_i \leftarrow c_i + 1$

(3) $c_i \leftarrow c_i - 1$

(4) if $c_i = 0$ then goto L_1 else goto L_2

(5) HALT(c_i) .

Such programs can simulate the computation of a Turing machine on a blank tape and hence their halting and divergence is unsolvable. Now, given a two-counter program, we construct a corresponding four-variable schema with variables y_1 , y_2 , y_3 , y_4 such that the schema halts if the program halts, and the schema diverges if the program does not halt (note: we will use reject statements as before). The statements corresponding to (1)-(5) above are

(1) START $\langle y_1, y_2, y_3, y_4 \rangle \leftarrow \langle a,a,a,a \rangle$

(2) $y_3 \leftarrow f(y_i)$;
if $y_3 = y_i$ then REJECT;
$y_4 \leftarrow a$;
while $y_4 \neq y_i$ do if $y_4 = y_3$ then REJECT else $y_4 \leftarrow f(y_4)$;
$y_i \leftarrow y_3$;

(3) $y_3 \leftarrow a$;

if $y_3 \neq y_i$ then

begin

L: $y_4 \leftarrow f(y_3)$;

if $y_4 \neq y_i$ then begin $y_3 \leftarrow y_4$; goto L end;

$y_i \leftarrow y_3$;

end

(4) $y_3 \leftarrow a$;

if $y_3 = y_i$ then goto L_1 else goto L_2

(5) HALT(a) .

This demonstrates the unsolvability of the one-function schemas.

3.3 Commutativity and Invertibility

3.3.1 Introduction

We now consider some classes of semi-interpreted schemas in which some of the base functions are related. In particular, we consider one-variable monadic flowchart schemas for which the class of possible interpretations may be restricted by the following specifications:

(i) two functions may be specified to commute (unary functions f and g are said to commute if $f(g(x)) = g(f(x))$ for all x),

(ii) some function is invertible (a function f is invertible if there exists another function f^{-1} such that $f(f^{-1}(x)) = f^{-1}(f(x)) = x$ for all x).

Thus, for a schema S, if f and g are specified to commute, then all interpretations are not allowed for S; only those interpretations are allowed that satisfy the formula $\forall x f(g(x)) = g(f(x))$. For a consideration of the inclusion, equivalence, and isomorphism problems for such semi-interpreted schemas we will only relate two schemas if they are compatible, i.e., they have the same specifications about commutative and invertible functions.

We show that with either commutativity or invertibility alone, the decision problmes of one-variable schemas remain solvable, but with both commutativity and invertibility they become unsolvable: we also relate some of these results to the equivalence problem of multi-dimensional automata.

All the schemas to be described below have a single variable (y) and one zero-ary function a. All other functions and predicates are unary. Unless otherwise specified, statements are of the following types:

(1) START $y \leftarrow a$

(2) HALT(τ)

(3) LOOP

(4) $y \leftarrow f_i(y)$

(5) <u>if</u> $p_i(\tau)$ <u>then goto</u> L_1 <u>else goto</u> L_2

where f_i is a unary function, p_i is a unary predicate, $\tau(y)$ is an arbitrary term that may or may not contain the variable y, and L_1 and L_2 are arbitrary labels.

3.3.2 Schemas with Commutative and Invertible Functions

Consider the class $\mathcal{C}_1$ of monadic flowchart schemas defined as follows. A schema S in $\mathcal{C}_1$ contains one variable y, a zero-ary function a, and an arbitrary number of unary functions $f_1, f_2, \ldots$ and unary predicates $p_1, p_2, \ldots$. In addition, there is a set E of pairs of functions $\{f_i, f_j\}$ that are specified to commute. Thus, if $\{f_i, f_j\} \in E$ then for any interpretation for S and any element x in the domain of the interpretation we must have $f_i(f_j(x)) = f_j(f_i(x))$. We refer to $\mathcal{C}_1$ as the class of commutative schemas.

Theorem 3.5 (Solvability of $\mathcal{C}_1$)

The class of commutative schemas is solvable, that is, for the class $\mathcal{C}_1$

(a) the halting problem is solvable,

(b) the divergence problem is solvable,

(c) the equivalence problem is solvable,

(d) the inclusion problme is solvable,

(e) the isomorphism problem is solvable.

For proofs, see Section 3.3.4.

Next, consider the class $\mathcal{C}_2$ of monadic flowchart schemas defined as follows. A schema S in $\mathcal{C}_2$ contains one variable y, a zero-ary function a, and unary functions $f^{-1}, f, f_1, f_2, \ldots$ and unary predicates $p_1, p_2, \ldots$, where f and f^{-1} are specified to be inverses, that is, for any interpretation for S, and any element x in the domain of the interpretation, we must have $f(f^{-1}(x)) = f^{-1}(f(x)) = x$.

Theorem 3.6 (Solvability of $\mathcal{C}_2$)

The class $\mathcal{C}_2$ of schemas with an invertible function is solvable.

For the proof, see Section 3.3.4.

Finally, consider the class of schemas that have both the commutativity and invertibility constraints. We wish to show that the decision problems for this class is unsolvable. For this, we exhibit the class $\mathcal{C}_3$ of periscopic schemas defined as follows (we call these schemas "periscopic" schemas because of their obvious relation to periscopic automata introduced in Section 3.3). A schema S in $\mathcal{C}_3$ has one variable y, one unary predicate p, the zero-ary function a, and three unary functions f^{-1}, f, g that are related by:

$$\forall x \; f(f^{-1}(x)) = f^{-1}(f(x)) = x$$

and

$$\forall x \; f(g(x)) = g(f(x)) \quad .$$

Note: this also implies that the functions f^{-1} and g commute. Tests in S have either the form p(y) or p(g(y)), and we also restrict halt statements to have the form HALT(a).

Theorem 3.7 (Unsolvability of $\mathcal{C}_3$)

Periscopic schemas are unsolvable. In other words, for $\mathcal{C}_3$

(a) the halting problem is unsolvable,

(b) the divergence problem is not partially solvable,

(c) the equivalence problem is not partially solvable,

(d) the inclusion problem is not partially solvable,

(e) the isomorphism problem is not partially solvable.

A question raised by this theorem is whether tests of the form $p(g(y))$ are really necessary for making the class $\mathcal{C}_3$ unsolvable. We might ask, for example, whether periscopic schemas without tests $p(g(y))$ might be solvable. The next theorem says that this is indeed the case.

Consider the class $\mathcal{C}_4$ of schemas which is like $\mathcal{C}_3$ except that the only tests allowed are of the form $p(y)$.

Theorem 3.8 (Solvability of $\mathcal{C}_4$)

The class $\mathcal{C}_4$ is solvable.

3.3.3 Application to Finite Automata Theory

From the above solvability and unsolvability results we wish similar results for finite automata. In general, the input tape of the automata we consider will be an infinite n-dimensional tape (with a root, or origin). We consider classes of automata by restricting the kinds of input tapes allowed and the possible ways the reading head of the automaton can move. An automaton may accept or reject its input tape, or it may run forever, in which case the tape is rejected.

Note that for automata we can consider the problems of acceptance, rejection, equivalence, inclusion and isomorphism as analogous to the problems of halting, divergence, equivalence, inclusion and isomorphism for schemas. The acceptance (rejection) problem is to decide if an automaton accepts (rejects) all input tapes, an automaton A_1 includes an automaton A_2 if the set of tapes accepted by A_1 contains all tapes

accepted by A_2, two automata are equivalent if they accept exactly the same set of input tapes, and two automata are isomorphic if for every input tape they "visit" and read exactly the same squares of the tape in the same order. We say that a class of automata is solvable if all these these problems are solvable for the class.

Schemas in $\mathcal{C}_1$ are closely related to finite automata on n-dimensional infinite tapes. An n-dimensional automaton is a finite state machine with one reading head that is initially at the "origin" of its n-dimension infinite tape. The symbols of the tape are from some finite alphabet Σ. The reading head of the automaton can, however, move only in the positive direction along any dimension. The automaton may halt and accept or reject the tape, or it may never halt (in which case the tape is rejected). We will represent the transition graph of the automaton by a program which has statements of the following kinds:

(1) L_0: START, goto $\delta(L_0,\sigma)$

(2) L_i: ACCEPT

(3) L_i: REJECT

(4) L_i: move(j), goto $\delta(L_i,\sigma)$

where move(j) means "move one step in the j-direction", and δ is a function from labels and tape symbols to labels -- σ stands for the symbol read from the tape (which is an element of Σ), and no $\delta(L_i,\sigma)$ can ever be the label L_0 for the start statement.

From Theorem 3.5 we obtain

Corollary A. The class of n-dimensional automata is solvable.

To show this we construct for every n-dimensional automaton A a corresponding schema $S \in \mathcal{C}_1$ (of Theorem 3.5). It will be obvious that

the acceptance, rejection, equivalence, inclusion, and isomorphism problems for n-dimensional automata are the same as the halting, divergence, equivalence, inclusion, and isomorphism problems for the corresponding schemas.

Given an n-dimensional automaton A on $\Sigma = \{\sigma_1,\ldots,\sigma_m\}$, we construct the corresponding schema $S \in \mathcal{C}_1$ as follows. S has n unary functions $f_1,\ldots,f_n$, each pair of which commutes, and $(m-1)$ unary predicates $p_1,\ldots,p_{m-1}$. Statements in the automaton A and the schema S correspond as follows:

Automaton A	Schema S
START	START $y \leftarrow a$
L_i: ACCEPT	L_i: HALT(a)
L_i: REJECT	L_i: LOOP
L_i: move(j), goto $\delta(L_i,\sigma)$	L_i: $y \leftarrow f_j(y)$; if $p_1(y)$ then goto $\delta(L_i,\sigma_1)$ else if $p_2(y)$ then goto $\delta(L_i,\sigma_2)$ ⋮ else if $p_{m-1}(y)$ then goto $\delta(L_i,\sigma_{m-1})$ else goto $\delta(L_i,\sigma_m)$

The head of the automaton corresponds to the variable y of the schema, the input tape for A corresponds to the interpretation for S, moving the head in direction j corresponds to applying the function f_j, and acceptance or rejection in A corresponds to halting or divergence in S. Note that for an input tape for A there correspond several interpretations for S, but it is obvious that the decision problems for the automata

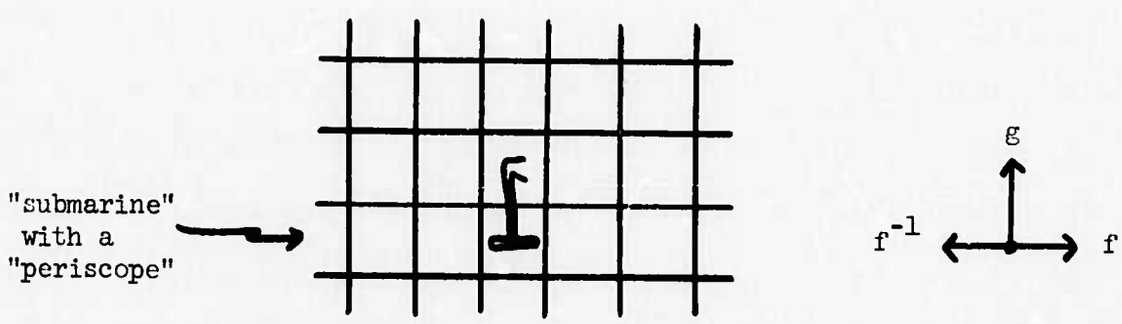
are reduced to the decision those for schemas (see also the canonical interpretations for $\mathcal{C}_1$ in the proof of Theorem 3.5).

It is clear that two-way finite state automata (on linear two-way infinite tapes) are related to schemas in $\mathcal{C}_2$ with unary functions f, f^{-1} in the same way as n-dimensional automata are related to schemas in $\mathcal{C}_1$. It follows, then, that

Corollary B. The class of two-way automata on one-dimensional infinite tapes is solvable.

Of course, this result is not new, but we mention it to show that it is derivable in a straightforward way from Theorem 3.6.

As we have done for classes $\mathcal{C}_1$ and $\mathcal{C}_2$, we describe a class of automata related to $\mathcal{C}_3$ that we call periscopic automata. A periscopic automaton has one head which can move on a two-dimensional infinite tape. We call the dimensions "horizontal" and "vertical". The head can move freely in the horizontal direction (i.e., left or right), but vertically it can move only upwards. However, attached to the head is a little "periscope" so that the automaton can read the symbol just above the head without moving the head vertically up. For our purposes it suffices to take the input alphabet to be of size two (we may say $\Sigma = \{T,F\}$).



The relation between a schema $S \in \mathcal{C}_3$ and the corresponding periscopic automaton A is obvious. An interpretation for S corresponds to an input tape for A, application of the functions f, f^{-1} and g in S correspond to moving the head of A right, left, and up respectively. It is the test p(g(y)) in S that gives the automaton A its periscopic vision. It is then easy to see from Theorem 3.7 that

Corollary C. The class of periscopic automata is unsolvable.

It is clear from this (and the proof of the theorem) that if we provide the automaton with any kind of periscope at all, e.g., arbitrarily high, inclined, or even pointing downwards, but not just horizontal, (for that is equivalent to no periscope at all), then the problems for the automata all remain unsolvable (and similarly for the corresponding schemas).

We say a periscopic automaton has periscopic vision if at least in one state it tests the symbol at the periscope. An automaton without periscopic vision is just an automaton that can move left, right and up, but not down, and can only look at the symbol under its reading head.

Theorem 3.8 shows that the decision problems for such automata are solvable.

Corollary D. The class of automata without periscopic vision is solvable.

3.3.4 Proofs

3.3.4.1 Proof of Theorem 3.5

We first give a proof of the solvability of the inclusion problem for a subclass $\mathcal{C}_1'$ of $\mathcal{C}_1$ in which any schema contains just two functions f_1, f_2 that commute, and one predicate p for which the only tests allowed are of the form $p(y)$, and halt statements have the form HALT(y). We will then give the proof of the solvability of $\mathcal{C}_1$, which will be on lines similar to the first proof.

Proof for $\mathcal{C}_1'$: We sketch the proof for the inclusion problem. Given two schemas S_1 and S_2 of $\mathcal{C}_1'$, to decide if $S_1 \leq S_2$. Now, without loss of generality we can assume that both S_1 and S_2 are free, for if they are not, they can trivially be made free. We also assume that from each assignment statement in S_1 and S_2, a halt statement can be reached, for otherwise we can replace such a statement by a loop statement.

Consider the class $\mathcal{I}$ of interpretations of the following kind. The domain of the interpretation is the set of strings $\{F_1^i F_2^j \mid i,j \geq 0\} \subset \{F_1,F_2\}^*$. The functions a, f_1, f_2 are defined as follows:

a is $F_1^0 F_2^0 = \Lambda$

$f_1(F_1^i F_2^j)$ is $F_1^{i+1} F_2^j$

$f_2(F_1^i F_2^j)$ is $F_1^i F_2^{j+1}$.

The predicate p is arbitrary.

Interpretations $\mathcal{I}$ play the same role for the class $\mathcal{C}_1'$ that Herbrand interpretations play for Herbrand schemas. If we associate with any interpretation I' an interpretation $I \in \mathcal{I}$ such that $p(F_1^i F_2^j)$ is true in I if and only if $p(f_1^i f_2^j(a))$ is true in I', and consider the homomorphism $\theta: I \to I'$ mapping $F_1^i F_2^j$ into the element $f_1^i f_2^j(a)$ of I' -- note, by the commutativity of f_1, f_2 this map is <u>onto</u> the reachable elements of I' (that is, elements that can be expressed as constant terms). Then, if we consider the computation of a schema $S \in \mathcal{C}_1'$ under I and I', they go through exactly the same sequence of statements of S, and the values of the variable correspond (under θ) at each step.

We can show that $S_1 \leq S_2$ if and only if $S_1 \leq S_2$ for the interpretations in $\mathcal{I}$. The "only if" part is trivial. For the "if" part, suppose $S_1 \not\leq S_2$. Then, for some interpretation I', S_1 halts, and S_2 either loops or halts with a different value. Then, if we consider the computations of S_1 and S_2 under the interpretation $I \in \mathcal{I}$ corresponding to I', we see that S_1 halts, and S_2 either loops or halts with a <u>different</u> value (by the existence of the homomorphism $\theta: I \to I'$). Thus $S_1 \not\leq S_2$ for the set of interpretations $\mathcal{I}$.

Now, given two schemas $S_1, S_2 \in \mathcal{C}_1'$, to decide if $S_1 \leq S_2$ we decide if $S_1 \leq S_2$ for the set of interpretations $\mathcal{I}$. We construct a finite state automaton A that simulates the computations of both S_1 and S_2 (in step) for an interpretation $I \in \mathcal{I}$ represented by the input tape of A. The tape consists of two tracks, one for each schema, and symbols on each track are from the set {T,F} representing the value of the predicate p applied to the current value of the variable y. It is the responsibility

of the automaton to detect whether or not the tape represents a feasible interpretation. At any instant in the computations of S_1 and S_2, let the values of the variable y be $F_1^{i_1} F_2^{j_1}$ in S_1, and $F_1^{i_2} F_2^{j_2}$ in S_2 (since the schemas are in step $i_1+j_1 = i_2+j_2$). Let the count c denote i_1-i_2. If the count is zero, the predicate p must have the same value on both tracks, else the values on the tracks may be arbitrary. The automaton A accepts an input tape unless S_1 halts and S_2 does not halt with the same output for the interpretation represented by the tape. Thus, the inclusion problem is reduced to the problem of deciding if a finite state automaton accepts all input tapes.

In its finite memory the automaton retains the following data:

(i) the current (assignment) statement executed by S_1, and by S_2, and

(ii) the value of the count c provided $|c| \leq \min(s_1, s_2)$ where s_1, s_2 are the number of assignment statements in S_1, S_2.

The automaton operates as follows:

(1) Read the input tracks (if the end-of-file is read, accept the tape). If $c = 0$ and the tracks read (T,F) or (F,T) then accept the tape ("impossible" interpretation).

(2) Using the values of p(y) from the tracks, "find" the next statements (other than test statements) for both schemas.

(3) If the next statement for S_1 is a halt statement then reject the tape unless $c = 0$ and S_2 also halts. If S_1 loops then accept the tape.

(4) If S_2 halts or loops on the next statement, reject the tape because as S_1 is free (over interpretations in $\mathcal{I}$) it can be made to reach a halt statement -- and it will apply at least one more function letter, thereby giving a different output from that of S_2.

(5) (Both next statements are assignment statements.) If S_1 executes $y \leftarrow f_1(y)$ and S_2 executes $y \leftarrow f_2(y)$ then increment c by 1; if S_1 executes $y \leftarrow f_2(y)$ and S_2 executes $y \leftarrow f_1(y)$ then decrement c by 1; otherwise leave c unchanged. If the new value of $|c|$ exceeds $\min(s_1, s_2)$ then reject the tape, otherwise, go to (1).

The reason that the input tape can be rejected if $|c|$ exceeds $\min(s_1, s_2)$ is that because S_1 and S_2 are free and "independent" for the next c steps, they can both reach halt statements without executing any statement twice (for some interpretation) -- and, of course, the outputs can be equal only if both reach halts at the same time and $c = 0$, but that is impossible because c changes by at most one in each step.

This completes the proof of the solvability of the inclusion problem (and hence also of the halting, divergence and equivalence problems) for $\mathcal{C}_1'$.

Proof for $\mathcal{C}_1$. The solvability of the halting and divergence are trivial because schemas in $\mathcal{C}_1$ can be made free. This can be done by making many copies of the schema, one for each partial specification state (see the notation in 3.2.4). A partial specification state

for schemas in $\mathcal{C}_1$ is a mapping from the set of atomic terms $p(\tau)$ such that $|\tau| \leq k$, into {true,false,unknown} provided it is consistent, i.e., it obeys commutativity relations, and if the value of y is $\tau()$, then $|\tau()| > k$ (for the initial part where $|\tau()| \leq k$, computation is done by expanding the schema out as a tree).

The solvability of equivalence follows from the solvability of inclusion (below).

For the proof of inclusion $(S_1 \leq S_2)$ we proceed as before by constructing an automaton A that accepts its input tape unless S_1 halts and S_2 does not halt with the same value.

First we describe the canonical interpretations for the schemas. Given S_1 and S_2 over unary functions $f_1,\ldots,f_n$ and predicates $p_1,\ldots,p_m$ and a set E of pairs of function symbols that commute, we define a class $\mathcal{I}$ of interpretations as follows. We define an equivalence relation on strings on $\Sigma = \{F_1,\ldots,F_n\}$ by the transitive closure of: $x_1,x_2 \in \Sigma^*$, $x_1 \equiv x_2$ if $x_1 = x_2$, or there exist $i,j \leq n$ such that $\{f_i,f_j\} \in E$ and x_2 can be obtained from x_1 by interchanging an occurrence of F_i with an adjacent occurrence of F_j. The domain of an interpretation $I \in \mathcal{I}$ is the set of equivalence classes of strings of Σ^* (an equivalence class is denoted by $\{x\}$ where x is a string in the class). The value of the function constant a is $\{\Lambda\}$, and functions $f_1,\ldots,f_n$ are defined in the obvious way, that is

$$f_i(\{x\}) = \{F_i.x\}$$

where the dot $(.)$ means the operation of concatenation, and the predicates $p_1,\ldots,p_m$ are arbitrary.

We note the following property of the domain of the interpretation

(*) $F_i \in \Sigma$ and $x,y \in \Sigma^*$, then $x.F_i \equiv y.F_i$ if and only if $x \equiv y$.

The "if" part is trivial. For the "only if" part, assume $x.F_i \equiv y.F_i$ and trace the position of the "rightmost" F_i as $x.F_i$ as transformed to $y.F_i$ by interchanging symbols (which correspond to pairs that are elements of E):

$$x.F_i = x_1 \to x_2 \to \ldots \to x_r = y.F_i \quad .$$

Now, if we consider $x_1', x_2', \ldots, x_r'$ where x_i' is the same as x_i, but with the rightmost F_i removed. Now it is easy to see that

$$x = x_1' \to x_2' \to \ldots \to x_r' = y$$

that is, $x \equiv y$. This completes the proof of the property (*).

Also, on lines very similar to the proof of C_1' we see that $S_1 \leq S_2$ if and only if $S_1 \leq S_2$ for the interpretations of $\mathcal{I}$.

We can now describe the automaton A. Let k denote $\max\{|\tau|\}$ of all terms τ used in S_1 and S_2. Now, a symbol on a track of the input tape gives the values of all $p_i(\tau)$ for all τ such that $|\tau| = k$ for S_1 and S_2. At any point in the simultaneous computations of S_1 and S_2, let the variables y in S_1 and S_2 have values $\{y_1\}$ and $\{y_2\}$, $y_1 = F_{i_1} F_{i_2} \ldots F_{i_r}$, and $y_2 = F_{j_1} F_{j_2} \ldots F_{j_r}$. Then we define the "unsaturated strings" x_1, x_2 of S_1, S_2 as follows: set $x_1 \leftarrow y_1$, $x_2 \leftarrow y_2$. Find the rightmost symbol F_i in x_1 that is common to both x_1 and x_2 (if one exists), say $x_1 = x_1' F_i x_1''$, $x_2 = x_2' F_i x_2''$, then if F_i commutes with each symbol in x_1'' and in x_2'' then set $x_1 \leftarrow x_1' x_1''$, $x_2 \leftarrow x_2' x_2''$, and repeat this process.

We describe the proof for the case where halts are of the form HALT(y). The general case HALT(τ) is easy to incorporate into the proof.

Since the schemas are free, any statement from which a halt cannot be reached is replaced by the loop statement.

In its finite control the automaton remembers

(i) the current (assignment) statement executed by S_1 and S_2,

(ii) for both S_1 and S_2, the values of all $p_i(\tau)$ for all non-constant terms τ such that $|\tau| \leq k-1$, and for all constant terms $\tau()$ such that $|\tau()| \leq k$, and

(iii) unsaturated strings $x_1, x_2 \in \Sigma$ such that x_1, x_2 have no symbol in common and $|x_1| = |x_2| \leq \min(s_1, s_2)+k$ where s_1, s_2 are the number of assignment statements in S_1, S_2.

From the property (*) we see that the values of the variable y in S_1 and S_2 are equal if and only if the unsaturated strings x_1, x_2 are both Λ. If there is some symbol common to both x_1, x_2 then we can show that the values of y in S_1 and S_2 have diverged, never to come together again. To show this, let F_i be the rightmost such symbol in x_1, and suppose it is "pushed" as much to the right in both x_1 and x_2 as possible. If it cannot reach the right end of x_1 (modified) then the modified x_1, x_2 have the form

x_1 is $\ldots F_i F_j \ldots$

x_2 is $\ldots F_i \ldots$

where F_i, F_j do not commute ($\{f_i, f_j\} \notin E$) and F_j does not occur to the right of F_i in x_2. Then, by extending x_1, x_2 to the left we cannot make them equivalent for the order of the rightmost F_i and F_j must be reversed in the two. On the other hand, if F_i cannot reach the right end in x_2 we have a similar argument. Hence if such a condition occurs the automaton rejects the input tape.

After observing this, we see that the lengths of the unsaturated strings ($|x_1| = |x_2|$) can change by at most one in any step, and if $|x_1| = c > k$ then the two schemas are "independent" at least for the next $(c-k)$ steps, so that if c exceeds $\min(s_1, s_2)+k$ the automaton can reject the input tape (see the argument in the proof for $\mathcal{C}_1'$).

We use the specification state approach of Section 3.2.4. We note that the automaton can check for the consistency of the values of $p(\tau)$ (given on the input tape) for the two tracks using the same argument of unsaturated strings, and that halts of the form HALT($\tau()$) can be handled in a straightforward way; from which we conclude that the inclusion problem has been shown to be solvable.

The proof of the solvability of the isomorphism problem for $\mathcal{C}_1$ is similar to the above, except that it is much simpler since unsaturated strings can never be anything other than Λ for otherwise the schemas are not isomorphic.

3.3.4.2 Proof of Theorem 3.6

Schemas in class $\mathcal{C}_2$ have the flavor of two-way finite automata. Applying the function f corresponds to moving the head right, applying f^{-1} corresponds to moving it left. There are some differences, however.

(i) the "input tape" is two-way infinite,

(ii) the schema outputs values,

(iii) the schema can test predicates on terms, and there are functions other than just f and f^{-1} .

Nevertheless, a proof somewhat similar to that for a two-way automaton works.

Given two schemas S_1 and S_2 of $\mathcal{C}_2$ having functions $f, f^{-1}, f_1, \ldots, f_n$, define the class $\mathcal{I}$ of canonical interpretations for S_1 and S_2 as follows: the domain is the set of strings of $\Sigma^* = \{F, F^{-1}, F_1, \ldots, F_n\}^*$ for which symbols F and F^{-1} do not appear adjacent to each other. The predicates $p_1, \ldots, p_m$ are arbitrary. As in the previous section, $S_1 \leq S_2$ (respectively S_1 and S_2 are isomorphic, S_1 halts, S_1 diverges) if and only if $S_1 \leq S_2$ for interpretations of $\mathcal{I}$ (respectively S_1 and S_2 are isomorphic for $\mathcal{I}$, S_1 halts on $\mathcal{I}$, S_1 diverges on $\mathcal{I}$).

(a) Halting. Given a schema $S \in \mathcal{C}_2$, to decide if S halts, we construct a finite state automaton A that accepts all input tapes if and only if S halts. The automaton A simulates the computation of S on an interpretation (from $\mathcal{I}$) represented by the input tape. At any point in the computation there is a value v we call the "pivot" element -- it is an element of Σ^* whose first symbol is not F or F^{-1}. For any element x of Σ^*, the specification state (SS) of x for an interpretation is defined to be the values of all $p(\tau(x))$ for all terms $\tau(x)$ for which $|\tau(x)| \leq k$ where k is the depth of the largest term used in S. The incomplete specification state (ISS) is the values for all $|\tau(x)| \leq k-1$. The state vector is a label L_i (that is executed) along with an ISS. A symbol on the input tape of the automaton represents the specification states of a pair of elements. Consecutive symbols give the specification states for the pairs

(v,v) , $(F^{-1}.v,F.v)$, $((F^{-1})^2.v,F^2.v)$, $((F^{-1})^3.v,F^3.v)$ ...

where v is a pivot element) -- until the pivot element is changed (as determined by A). The first element of a pair is called the left element, the second the right element.

The first pivot element is Λ (corresponding to the function constant a). The automaton works as follows. It retains a table of "instances" and "outcomes". For both the left and the right value there is an instance of the variable y for each assignment statement L_i of S , which corresponds to the computation if L_i is exited with this value for y . In addition, there is one primary instance which corresponds to the real computation of the schema. Let $((F^{-1})^r.v,F^r.v)$, $r \geq 0$, be the current elements, with v as the pivot. The outcome for each instance can be one of five possibilities:

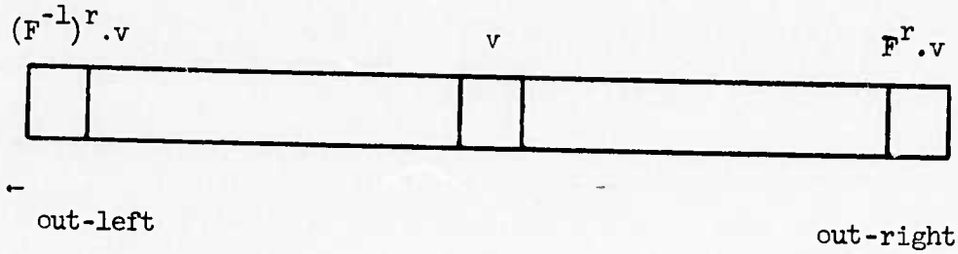
(1) halt,

(2) exit (with some state vector) -- it corresponds to an execution of an assignment $y \leftarrow f_i(y)$ (f_i is not f or f^{-1}),

(3) out-left (with some statement L_i) -- it corresponds to executing L_i: $y \leftarrow f^{-1}(y)$ where (the old) y had value $(F^{-1})^r.v$,

(4) out-right (with some statement L_i) -- it corresponds to L_i: $y \leftarrow f(y)$ where y had value $F^r.v$,

(5) diverge -- the computation for this instance either enters a loop statement, or diverges.



Instances	Outcomes
primary instance (real computation)	halt
	exit ISS
y_i left (statement L_i exited with value $(F^{-1})^r.v$)	out-left L_i
	out-right L_i
y_i right (statement L_i exited with value $F^r.v$)	diverge

In its finite memory the automaton has

(i) the current table of instances and outcomes,

(ii) the incomplete specification states (ISS) of the next pair to read in,

(iii) the value of r if $r \leq k$; and the value $\tau()$ of the pivot element v if $|\tau()| \leq k$.

We call (i) and (ii) the complete state of the schema. The schema also retains

(iv) all complete states entered for the current pivot element, and

(v) all state vectors for all pivot elements entered.

The reason for (iv) is that if the complete state repeats, the schema can be made to diverge with the primary instance making assignments only like $y \leftarrow f(y)$ and $y \leftarrow f^{-1}(y)$. The reason for (v) is that if the state vector for a pivot element repeats, the schema can be made to diverge because pivot elements are independent, i.e., all information regarding previous tests is "lost" (except the ISS) when an assignment like $y \leftarrow f_i(y)$ is made.

The automaton operates as follows:

(1) Read the specification state for the pivot element. The ISS part must match the required ISS (unless this is the first element -- Λ) -- if not, accept the tape, otherwise set up the required tables.

(2) If the primary instance halts -- accept the tape.
If the primary instance diverges -- reject the tape.
If the primary instance exits then we have a new pivot element -- if its ISS repeats, reject the tape, else go to step (1).
If the table repeats -- reject the tape.

(3) Read the next pair of predicate states. If it is an "impossible" interpretation, accept the tape, otherwise update the tables and go to step (2).

(b) Divergence. This can be proved like the halting problem, only the automaton is simpler. It does not need to remember the information (iv), (v); instead, it simply simulates the computation and rejects the input tape if the primary instance halts, and accepts it if the interpretation is "impossible", or the end-of-file is reached.

From the proof it follows that it is solvable whether or not a schema would always diverge when any new pivot element is entered with any specified state vector. This fact is used in the proof of inclusion below.

(c) Equivalence. The solvability of equivalence follows from the solvability of inclusion below.

(d) Inclusion. Given two schemas $S_1, S_2 \in \mathcal{C}_2$, to decide if $S_1 \leq S_2$, we construct an automaton A, similar to the automaton in part(a), such that A accepts all input tapes if and only if $S_1 \leq S_2$. The automaton simulates the computation of all instances of both schemas. The possible outcomes for each instance are

(1) halt, with some value x,

(2) exit, with some state vector and some value x -- it corresponds to an execution of $y \leftarrow f_i(y)$ where f_i is not f or f^{-1}, and x is the (old) value of y,

(3) out-left, with some statement L_i,

(4) out-right, with some statement L_i,

(5) diverge.

The automaton need not (and indeed cannot) remember the value x for all halt or exit outcomes; it suffices to remember the equivalence classes of outcomes that halt or exit with the same value, and the values of only those instances that halt with output $\tau()$, $|\tau()| \leq k$.

In its finite memory the automaton stores (as in the proof of halting):

(i) the table of instances and outcomes for both S_1 and S_2,

(ii) the incomplete specification state of the next pair,

(iii) the value of r, if $r \leq k$; and the value $\tau()$ of the pivot element v if $|\tau()| \leq k$.

In addition, the automaton has the capability of storing

(iv) an arbitrary set of complete states of S_2 (tables of instances and outcomes for S_2, and ISS of next pair). This is required in steps 2(iv) and 4(iv) below.

For simplicity we only show the proof for schemas in which a halt must have the form HALT(y). The general case HALT(τ) is easy to incorporate.

The automaton operates as follows. On seeing an end-of-file it accepts the tape. Otherwise it reads a pair of specification states from the tape, checks if they match with the known incomplete specification states. If not, the tape is accepted ("impossible" interpretation). If they match, then

(1) if the principal instance for schema S_1 diverges, then the tape is accepted,

(2) if S_1 halts then

  (i) if S_2 halts with the same value -- accept.

  (ii) if S_2 halts with a different value -- reject.

  (iii) if S_2 exits -- reject.

  (iv) if none of the above, then continue simulation of S_2 and construct the set of complete states until either (i), (ii) or (iii) above applies, or a complete state repeats -- in which case reject the tape,

(3) if S_1 exits in a state vector which must loop (decidable -- see the divergence problem) then accept the tape,

(4) if S_1 exits in a state vector from which it can halt, then

  (i) if S_2 halts, then reject,

  (ii) if S_2 exits with a different value, then reject,

(iii) if S_2 exits with the same value, continue simulation of both S_1, S_2,

(iv) if none of the above, then continue simulation of S_2, constructing the set of complete states until (i), (ii), or (iii) above apply, or a complete state repeats, in which case reject the tape,

(5) if none of the above, continue simulation of both S_1 and S_2.

This completes the proof.

(e) Isomorphism. An automaton is constructed as in case (d) above, except it also keeps track (in the table of instances and outcomes) which instances undergo isomorphic computations. Then, the automaton rejects a tape if the computations of the principal instances of both schemas are not isomorphic at any step.

3.3.4.3 Proof of Theorem 3.7

To show the unsolvability of schemas in $\mathcal{C}_3$, we reduce the halting problem for null-input Post machines to the halting and divergence problems for $\mathcal{C}_3$. A Post machine over {a,b} is a machine operating on strings, and having the following statements:

```
START(x)
HALT
LOOP
x ← x.a
x ← x.b
if x = Λ then goto L1
    else if head(x) = a then begin x ← tail(x); goto L2 end
                        else begin x ← tail(x); goto L3 end
```

where L_1, L_2, L_3 are arbitrary labels, and head(x) represents the first symbol of x, and tail(x) represents the rest of the string x.

Given a Post machine M we will construct a schema S which looks like a schema of $\mathcal{C}_3$ except it has special statements called reject statements. Replacing reject statements by halt statements gives us a schema that halts if and only if the machine M halts on input Λ, and replacing them by loop statements gives a schema that diverges if and only if M does not halt.

The idea is that any interpretation for S can be represented by a grid of integer nodes in a half plane (doubly infinite along the x-axis). The constant function a corresponds to the origin; applying the function f corresponds to moving right, applying f^{-1} corresponds to moving left, and applying g corresponds to moving up. At each node we have a T or F value, corresponding to the value of the predicate p (see the canonical interpretation for the class $\mathcal{C}_1'$ in Section 3.3.4.1).

		g(a)	gf(a)		
	$f^{-1}(a)$	a	f(a)	$f^2(a)$	

The schema S can simulate the computation of M on this plane as follows. It uses two horizontally adjacent nodes to "code" a letter (either a, b or e -- a special end marker: a corresponds to TT, b to TF, e to F-). In this manner, the schema will "lay off" a

current value of the string x (of M) in one row of nodes, enclosed by end-markers. The next string (after M executes one step) will be laid off on the next higher row. The schema S will simply check this computation. If the interpretation doesn't agree, the interpretation will be rejected.

In our schema S we will allow the use of predicate tests of the form $p(f(y))$, $p(g(f(y)))$, etc., since these can be implemented using only the allowed statements ($y \leftarrow f(y)$; $p(g(y))$; $y \leftarrow f^{-1}(y)$ for the test $p(g(f(y)))$, etc.). The correspondence between statements in M and those in S can be set up as follows.

We first define the macros

```
CHECK   = y ← ff(y);
          while p(y) do
             begin if p(y) ⊗ p(g(y)) then REJECT;
                   if p(f(y) ⊗ p(gf(y)) then REJECT;
                   y ← ff(y);
             end; comment ⊗ represents exclusive-or;
CHECKA  = if ¬ p(g(y)) then REJECT;
          if ¬ p(gf(y)) then REJECT;
CHECKB  = if ¬ p(g(y)) then REJECT;
          if p(gf(y)) then REJECT;
CHECKE  = if p(g(y)) then REJECT;
BACKUP  = y ← f^-1 f^-1 (y);
          while p(y) do y ← f^-1 f^-1 (y);
```

The correspondence between statements in M and those in S :

Statement in M	Statements in S
START(x)	START $y \leftarrow a$;
	if p(y) then REJECT;
	if p(ff(y)) then REJECT;
HALT	HALT(y)
LOOP	LOOP
$x \leftarrow x.a$	CHECKE; CHECK; CHECKA;
	$y \leftarrow ff(y)$; CHECKE;
	$y \leftarrow f^{-1}f^{-1}(y)$; BACKUP;
	$y \leftarrow g(y)$;
$x \leftarrow x.b$	CHECKE; CHECK; CHECKB;
	$y \leftarrow ff(y)$; CHECKE;
	$y \leftarrow f^{-1}f^{-1}(y)$; BACKUP;
	$y \leftarrow g(y)$;
if $x = \Lambda$ then goto L_1	if $\neg$ p(ff(y)) then goto L_1;
else if head(x) = a then	if p(fff(y)) then
begin $x \leftarrow tail(x)$;	begin $y \leftarrow ff(y)$; CHECKE;
goto L_2;	CHECK; CHECKE; BACKUP;
end	$y \leftarrow gff(y)$;
else begin $x \leftarrow tail(x)$;	goto L_2;
goto L_3;	end
end	else
	begin $y \leftarrow ff(y)$; CHECKE;
	CHECK; CHECKE; BACKUP;
	$y \leftarrow gff(y)$;
	goto L_3;
	end

This completes the proof of the unsolvability of the halting problem and the non-partial solvability of the divergence problem which in turn implies the non-partial solvability of equivalence, inclusion, and isomorphism.

3.3.4.4 Proof of Theorem 3.8

The main difference between a schema in $\mathcal{C}_4$ and a schema in $\mathcal{C}_3$ is that in $\mathcal{C}_4$, after an assignment statement $y \leftarrow g(y)$ the subsequent path of computation is completely independent of the outcomes of earlier predicate tests. For this reason, the proofs of the solvability of halting, divergence and isomorphism of $\mathcal{C}_2$ also work for $\mathcal{C}_4$.

The solvability of equivalence follows from the solvability of inclusion (below).

For the proof of inclusion we proceed along lines similar to the corresponding proof in $\mathcal{C}_2$. But, first we observe that any interpretation for schemas in $\mathcal{C}_4$ can be represented as a half plane (as in the case of $\mathcal{C}_3$). We use the notion of "distance" between two values, which denotes the horizontal distance between them on the plane. Secondly, from each statement $L_i: y \leftarrow g(y)$ of a schema S we can decide whether or not S must loop, and if not, we can find the shortest number of steps n_i in which S can be made to halt after executing L_i.

Now, given two schemas $S_1, S_2 \in \mathcal{C}_4$, to decide if $S_1 \leq S_2$, let c_1 denote $\max\{n_i\}$ for statements $L_i: y \leftarrow g(y)$ in S_1 from which S_1 can halt; and similarly c_2 is for S_2. We construct an automaton A that simulates the computations of S_1 and S_2 as in the proof for $\mathcal{C}_2$.

However, its table of instances and outcomes is somewhat different. It keeps track of the "distance" between those outcomes that exit provided the distance is no more than c_1+c_2 .

The rules for accepting/rejecting an input tape are as follows. If an end-of-file or an "impossible" interpretation is seen, the tape is accepted. Otherwise

(1) if the principal instance for schema S_1 diverges, then the tape is accepted,

(2) if S_1 halts then

  (i) if S_2 halts with the same value -- accept,

  (ii) if S_2 halts with a different value -- reject,

  (iii) if S_2 exits -- reject,

  (iv) if none of the above, then continue simulation of S_2 and construct the set of complete states until either (i), (ii) or (iii) above applies, or a complete state repeats -- in which case reject the tape,

(3) if S_1 exits in a state vector (since the incomplete specification state is null, the state vector consists simply of one label) from which S_1 must loop, then accept,

(4) if S_1 exits in a state vector from which it can halt, then

  (i) if S_2 halts, then reject,

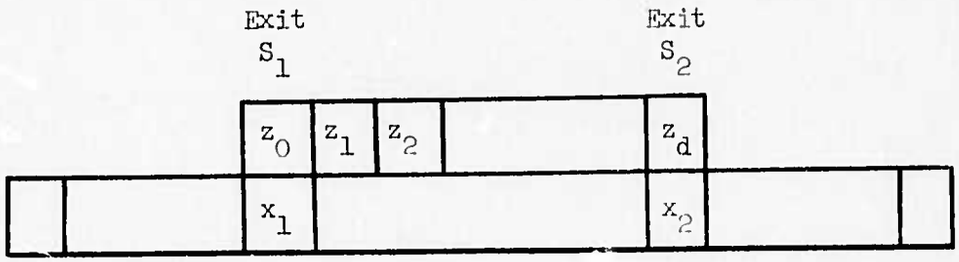
  (ii) if S_2 exits with a value more than c_1+c_2 distant, then reject,

  (iii) if S_2 exits with a value distant d from S_1 , $d \leq c_1+c_2$, then the next symbol read must be a "special symbol". If x_1 , x_2 are the values with which S_1 , S_2 exit, then we have a sequence of values

$$g(x_1) = z_0 \rightarrow z_1 \rightarrow \ldots \rightarrow z_d = g(x_2)$$

such that each $z_{i+1} = f(z_i)$ or each $z_{i+1} = f^{-1}(z_i)$. Then this special symbol provides the values of

$p(z_1), p(z_2), \ldots, p(z_{d-1})$.



The special symbol is used to set up the instance-outcome table again, and continue simulation,

(iv) if none of the above, then continue simulation of S_2 constructing the set of complete states until (i), (ii), or (iii) above apply, or a complete state repeats, in which case reject the input tape,

(v) if none of the above, then continue simulation of both S_1 and S_2 .

The justification for 4(ii) above is that if S_1 , S_2 exit more than $c_1 + c_2$ apart S_1 can be made to halt, and S_2 will either loop, or can independently be sent to halt statement with a different value (under some interpretation).

This completes the proof.

Chapter 4 Generalized Flowchart Schemas

4.1 Introduction

Ianov [1960] considered the data-space of a program to be representable by a single value, that could be changed (by applying a function) or tested (by a predicate). These base functions and predicates were assumed to be total, but otherwise completely uninterpreted. The idea was that by this mechanism one could model the control structure of computations and possibly even prove some useful properties about real programs, e.g., halting and equivalence. Unfortunately, the problem with this simple model was that two programs which computed the same value for all possible inputs but went about their task in slightly different ways were treated as being non-equivalent under this model -- we had lost too much information, firstly, by making the base functions and predicates totally uninterpreted, and secondly, by treating the whole of the data space as being a single element in the domain.

The latter objection was partially answered by Luckham, Park and Paterson [1970] when they treated the data space as consisting of a finite number of parts which could be manipulated by the program. While an improvement, this model too could not usefully represent computations in which memory requirements increase with the duration of the computation. Also, quite basic control features, e.g., markers were missing. Subsequently there have been several attempts to answer that latter objection by considering the subdivision of the memory into greater and even greater detail -- labels, label stacks, counters, markers, boolean variables, one- and many-dimension arrays, lists, etc.,

have been considered. These may be called structural features, and one can construct an endless number of these -- stacks of arrays, arrays of stacks, arrays with a dynamic number of arguments, general data structures like those of ALGOL 68, and so on. While it is true that most of these do not add any "inherent" power to the schemas, i.e., any schema in one class can be translated into an equivalent schema of another class, one cannot be completely satisfied with a "minimal" class since the aim of the study of schemas is to model computations, not just to obtain a machine capable of computing the partial recursive functionals. This is akin to the similar state of affairs for partial functions -- a three counter machine (that can increment, decrement, and test a counter for being zero) can compute all the partial recursive functions, and yet it is hardly a good model for computer programs.

Are we then arguing for a profusion of classes of schemas, one for each subset of possible data types, with little unifying theory? No. On the contrary, it would be quite useful to construct a rather general class of schemas from which many of the others can be obtained as subclasses.

While significant effort has been devoted by researchers to answer the second objection to Ianov's model, viz., the problem of a single data space, relatively little effort has been devoted towards the first objection, i.e., that one loses too much information in considering all the base functions and predicates to be uninterpreted. One would like to specify, for example, that two functions commute, or that a certain relation is transitive. In studies, most of these notions have not been integral parts of schemas in the discussion of properties of classes of

schemas, but they crop up, in an ad hoc way, when a specific schema is used to model a specific program.

It is our intention to handle these two basic problems in a uniform way, viz., by defining the class of generalized flowchart schemas. Generalized schemas have the inherently sound philosophy of Ianov that the complete data space of a program can be represented as a value (in some domain) but that operations on it may have the effect of modifying specific parts of the memory while leaving others unchanged. A generalized schema $S = \langle F,\varphi,P\rangle$ is a flowchart F (with a single variable), an attached formula φ of first order predicate calculus with equality, and a set P of function and predicate symbols, which corresponds to the set of base function and predicate symbols of the schema. The relevant interpretations for S are those that satisfy φ, not all possible interpretations (as in the case of totally uninterpreted base functions and predicates). We show that generalized schemas have the power of modelling the other classes of schemas, i.e., those that concentrate on the subdivision of memory. The other dilemma between the completely interpreted programs and the completely uninterpreted program schemas is satisfied by specifying as much or as little about the interpretation (by the formula φ) as may be desired for any specific application.

This chapter introduces the class of generalized flowchart schemas and shows some of the possibilities of modelling structural subdivisions of memory and other useful properties. We then show how most of the classical theory of schemas can be represented by these schemas, and finally we prove the fundamental theorem of maximal schemas that states that schemas with arrays and equality tests are, in some sense, a maximal class.

4.2 Definition of Generalized Schemas

4.2.1 Basic Definitions

In the rest of this chapter whenever we say "schema" we mean a generalized schema. Sometimes we also use the phrase φ-schema to mean a generalized schema. Schemas of the earlier chapters will be called conventional schemas.

A schema $S = \langle F,\varphi,P\rangle$ consists of a flowchart F, a formula φ of first order predicate calculus with equality and a finite set P of function and predicate symbols. The flowchart F has a very special form. There is only one variable (we call it y), and statements consist of the following:

Start statement	START $y \leftarrow \tau()$
Halt statement	HALT $(\tau(y))$
Loop statement	LOOP
Assignment statement	$y \leftarrow \tau(y)$
Test statement	if $\alpha(y)$ then goto L_1 else goto L_2 ,

where $\tau()$ represents a constant term, $\tau(y)$ represents any term, and $\alpha(y)$ represents any atomic formula, i.e., a predicate or equality test. For convenience we will use ALGOL-like notation instead of strict flowchart notation. We hence allow the use of labels and goto statements, with the tacit understanding that there exists no cycle consisting entirely of goto-statements.

An interpretation I for a schema $S = \langle F,\varphi,P\rangle$ is one that specifies at least the functions and predicates used in F, φ and P. But the only interpretations of interest are those that satisfy φ -- we write $I \models \varphi$ if the interpretation I satisfies φ, and we say that *I is an interpretation for S*.

If S is a schema and I is an interpretation for S, we use the notation $\underline{Dom(I)}$ to mean the domain of the interpretation, and $\underline{Val(S,I)}$ to mean the output of the computation of S on I. If S diverges on I then $Val(S,I)$ is undefined. Similarly, $\underline{Path(S,I)}$ is the path of the computation of S on I (for an exact definition of a path, see Section 2.1.4). Also, if $S = \langle F,\varphi,P\rangle$, we use the notation $\underline{\Sigma(S)}$ to denote the set of function and predicate symbols appearing in S, i.e., in F, φ, or in P.

Definition. Given an interpretation I on a domain $Dom(I)$ over a set of function and predicate symbols Q, we define the subinterpretation I' of I with respect to a set P of function and predicate symbols in the following way: the domain $Dom(I')$ of I' is the smallest subset of $Dom(I)$ closed under the functions in $P \cap Q$, and the values of the functions and predicates of $P \cap Q$ are the same in I' as in I. Note that if P does not contain any zero-ary function then the domain $Dom(I')$ is empty. We use the notation $\underline{I/P}$ to represent the subinterpretation of I with respect to P.

Definition. A schema $S = \langle F,\varphi,P\rangle$ is said to be well-founded if for every two interpretations I_1, I_2 for S (i.e., $I_1 \models \varphi$ and $I_2 \models \varphi$) such that there is an isomorphism θ from (I_1/P) to (I_2/P), then

(i) $Path(S,I_1) = Path(S,I_2)$, and

(ii) if the computations halt, then $Val(S,I_2) = \theta(Val(S,I_1))$.

The significance of a set P that makes $S = \langle F,\varphi,P\rangle$ well founded is that for any interpretation for S, knowledge of merely the functions

and predicates P is sufficient to characterize the computation. Given F and φ, a minimal set P for which $\langle F,\varphi,P\rangle$ is well founded represents the minimal set of functions and predicates whose values are sufficient to fully characterize a computation. If only the values of a smaller set of functions and predicates are fixed, then there is some indeterminacy as to what the schema will do, i.e., there are two interpretations both of which satisfy φ, and also agree over the fixed values, but the paths of the computations on I_1 and I_2 are different, or the outputs are different.

We will only be interested in schemas that are well founded, and in the rest of this chapter, all schemas considered are well founded unless otherwise specified.

It should be noted that if $S = \langle F,\varphi,P\rangle$ is well founded and I_1, I_2 are interpretations for S whose subinterpretations with respect to P are isomorphic, then (a) if the computation of S on I_1 halts then its computation on I_2 also halts after exactly the same number of steps, and (b) the outputs of the two computations $Val(S_1,I_1)$ and $Val(S_2,I_2)$ are elements of $Dom(I_1/P)$ and $Dom(I_2/P)$ respectively.

It follows from the definition that

(a) given any F and φ, if we let Q denote the set of function and predicate symbols in F, then $\langle F,\varphi,Q\rangle$ is well founded.

(b) if $\langle F,\varphi,P\rangle$ is well founded, and Q is any set such that $P \subset Q$, then $\langle F,\varphi,Q\rangle$ is also well founded, and

(c) if φ is "false", then $\langle F,\varphi,P\rangle$ is well founded for all F and P.

It is also easy to see that in general it is not partially solvable whether a schema S is well founded. This follows directly from the

fact (intuitively plausible to all schematologists, and proved in Section 4.5) that the divergence problem for φ-schemas is not partially solvable. The unsolvability of well foundedness should not shock us unduly. The corresponding problem for conventional schemas, too, is not partially solvable. For, consider a conventional schema S with a statement HALT(b) where b is a zero-ary function not used in the rest of S. Now we ask if the computation of S can be specified if we give an interpretation for S, but refuse to specify the value of the zero-ary function b. If the HALT(b) statement happens to be disconnected from the rest of S, the answer is yes, but in general it is unsolvable.

The correspondence between conventional schemas and generalized schemas can be represented by the following table.

Conventional schema	φ-schema
The total data space	The variable y
Functions and predicates	The set P
Interpretation	(I/P)
The structure of the data space, and totally interpreted features (like counters)	Predicates and functions other than those in P, related by the formula φ.

This also shows why we are interested only in the well founded schemas; for, in a conventional schema, if we specify only the values of a subset of the base functions and predicates, it may not be adequate to characterize the computation, and this represents an "incompleteness" in the schema.

A schema $S = \langle F,\varphi,P\rangle$ halts for an interpretation I if the computation of the flowchart F under I reaches a halt statement.

A schema $\langle F,\varphi,P\rangle$ is said to halt if it halts for every interpretation I for S (i.e., $I \models \varphi$). Similarly, a schema is said to diverge if for every I for S the schema does not halt. A schema S is free if for every path K in S there is an interpretation I for S such that $K = \text{Path}(S,I)$.

In the special case where φ is "false", the useless schema $\langle F, \text{false}, P\rangle$ both halts and diverges as there is no I for which $I \models \text{false}$. In the other special case where φ is "true" the schemas so obtained are the conventional one-variable schemas, i.e., $\mathcal{C}$(1 var) -- these are very similar to the Ianov schemas except that in Ianov schemas the assignments and tests are somewhat simpler.

This describes the class of generalized schemas. We can take interesting subclasses of these schemas by restricting the kinds of flowcharts and the formulas φ allowed. In fact, by specifying φ we can obtain schemas that behave as if the schemas had several variables (conventional n-variable schemas), or counters, or pushdown stacks, or other structural features. In each case, however, the single variable y corresponds to the entire data space of the schema. We will consider this aspect in Section 4.4.

4.2.2 Some Examples

We now give some simple examples of generalized schemas.

Example 1

Consider the schema $S_a = \langle F_a,\varphi_a,P_a\rangle$. There are two zero-ary functions a_0, a_1 and two binary functions f_+, f. . The formula φ_a is:

$$
\begin{aligned}
& a_0 \neq a_1 \\
& \wedge \forall x \forall y f_+(x,y) = f_+(y,x) \wedge f.(x,y) = f.(y,x) \\
& \wedge \forall x \forall y \forall z f_+(f_+(x,y),z) = f_+(x,f_+(y,z)) \wedge f.(f.(x,y),z) = f.(x,f.(y,z)) \\
& \wedge \forall x \quad f_+(x,a_0) = x \wedge f.(x,a_1) = x \\
& \wedge \forall x \exists y \quad f_+(x,y) = a_0 \\
& \wedge \forall x \quad (x \neq a_0) \rightarrow \exists y f.(x,y) = a_1 \\
& \wedge \forall x \forall y \forall z f.(x,f_+(y,z)) = f_+(f.(x,y),f.(x,z)) \\
& \qquad \wedge f.(f_+(x,y),z) = f_+(f.(x,z),f.(y,z)) \quad .
\end{aligned}
$$

The flowchart F_a is:

START $y \leftarrow a_1$;

<u>while</u> $y \neq a_0$ <u>do</u> $y \leftarrow f_+(y,a_1)$;

HALT (a_0) ,

and the set P_a is $\{a_1,f_+\}$.

An interpretation for the schema S_a is a commutative field. The schema halts if and only if the characteristic of the field is finite. Note that the zero-ary function a_0 is not in P_a , but the schema is well founded.

<u>Example 2</u>

Consider the schema $S_b = \langle F_b,\Phi_b,P_b\rangle$. S_b has one zero-ary function a , three unary functions f , car , cdr , one binary function cons , and one unary predicate p .

Φ_b is $\forall x \forall y\, car(cons(x,y)) = x \wedge cdr(cons(x,y)) = y$,

F_b is START $y \leftarrow cons(a,cons(f(a),a))$;

L_1: $y \leftarrow cons(ff(car(cdr(y))),y)$;

if $p(car(y))$ then HALT$(car(y))$;

$y \leftarrow cons(f(car(cdr(y))),y)$;

if $\neg\, p(car(y))$ then HALT$(car(y))$;

goto L_1 ,

and

P_b is $\{a,f,p\}$.

The schema halts. In fact, the output of S_c on any interpretation I can be given by the following formula:

$$Val(S_b,I) = \text{if } p(f^3(a)) \text{ then } f^3(a)$$
$$\text{else if } \neg\, p(f(a)) \text{ then } f(a)$$
$$\text{else if } p(f^5(a)) \text{ then } f^5(a)$$
$$\text{else if } \neg\, p(f^2(a)) \text{ then } f^2(a)$$
$$\text{else if } p(f^7(a)) \text{ then } f^7(a)$$
$$\text{else } f^3(a) \quad .$$

The notion of the equivalence of the two schemas will be defined in the next section but intuitively the schema S_b is "equivalent", in some sense, to the schema $S_c = \langle F_c,\Phi_c,P_c\rangle$ defined below (we use the abbreviation $f^2(a)$ for $ff(a)$, etc.):

Ψ_c is true

F_c is START y ← a;

if $p(f^3(a))$ then HALT($f^3(a)$);

if $\neg\, p(f(a))$ then HALT($f(a)$);

if $p(f^5(a))$ then HALT($f^5(a)$);

if $\neg\, p(f^2(a))$ then HALT($f^2(a)$);

if $p(f^7(a))$ then HALT($f^7(a)$);

HALT($f^3(a)$) ,

and

P_c is {a,f,p} , i.e., the same as P_b .

4.3 Equivalence of Schemas

4.3.1 Introduction

What does it mean to say that two schemas S_1 and S_2 are equivalent? Saying S_1 and S_2 are equivalent means that the outputs of S_1 and S_2 should be the same if both schemas are made to compute on the same interpretation. However, there is one point that this simple notion overlooks. It is that all relevant interpretations for the first schema need not be the same as all the relevant interpretations for the second schema, as in the case of Example 2 in the previous section where the functions car , cdr and cons represented structural features in S_b which were absent in S_c . The values in the domain of an interpretation for a schema represent the data space of the schema, and correspond to both the structural and the non-structural aspects. However, it is only the non-structural aspects that are crucial for the definition of

equivalence. It is precisely this dichotomy between the structural and the interpretive aspects of a schema that dictates a little care in the definition of equivalence. This problem does not arise in conventional schemata theory because these two aspects of schemas are well segregated, and it is because we wish to give a unified treatment that we are forced to confront the issue.

4.3.2 Definitions

We remark again that all schemas considered below are assumed to be well founded.

Definition. We say that two schemas $S_1 = \langle F_1, \Phi_1, P_1 \rangle$ and $S_2 = \langle F_2, \Phi_2, P_2 \rangle$ are compatible if $P_1 = P_2$.

Definition. $S_2 = \langle F_2, \Phi_2, P \rangle$ is a generalization of $S_1 = \langle F_1, \Phi_1, P \rangle$ if:

$\forall I_1$ for S_1 i.e., $I_1 \models \Phi_1$

$\exists I_2$ for S_2 i.e., $I_2 \models \Phi_2$ and

$\exists$ an isomorphism θ: $(I_1/P) \leftrightarrow (I_2/P)$

such that if S_1 halts on I_1 then S_2 also halts on I_2 and $Val(S_2, I_2) = \theta(Val(S_1, I_1))$; and if $Val(S_1, I_1)$ is undefined then $Val(S_2, I_2)$ is also undefined.

If S_2 is a generalization of S_1 we write $S_1 \underset{gen}{\leq} S_2$.

Note that the definition of well foundedness implies that for any interpretation I_1 for S_1, if there exist two interpretations I_2, I_3 for S_2 whose subinterpretations over P are isomorphic to (I_1/P), i.e.,

Θ_2: $(I_1/P) \leftrightarrow (I_2/P)$

and Θ_3: $(I_1/P) \leftrightarrow (I_3/P)$,

then if $Val(S_2,I_2) = \Theta_2(Val(S_1,I_1)$

then $Val(S_2,I_2) = \Theta_3(Val(S_2,I_2)$.

It is clear from the definition that generalization is reflexive and transitive.

Definition. $S_2 = \langle F_2,\Phi_2,P\rangle$ includes (is at least as defined as) $S_1 = \langle F_1,\Phi_1,P\rangle$ if:

(i) $\forall I_1$ for S_1 , $\exists I_2$ for S_2 and $\exists$ an isomorphism Θ: $(I_1/P) \leftrightarrow (I_2/P)$ such that if S_1 halts on I_1 then S_2 also halts on I_2 , and $Val(S_2,I_2) = \Theta(Val(S_1,I_1))$, and

(ii) $\forall I_2$ for S_2 , $\exists I_1$ for S_1 and $\exists$ an isomorphism Θ: $(I_1/P) \rightarrow (I_2/P)$ such that if S_1 halts on I_1 then S_2 also halts on I_2 , and $Val(S_2,I_2) = \Theta(Val(S_1,I_1))$.

If S_2 includes S_1 we write $S_1 \leq S_2$.

Definition. We say that two compatible schemas S_1 and S_2 are equivalent $(S_1 \equiv S_2)$ if $S_1 \underset{gen}{\leq} S_2$, and $S_2 \underset{gen}{\leq} S_1$.

Alternatively, $S_1 \equiv S_2$ if and only if $S_1 \leq S_2$, and $S_2 \leq S_1$.

We should now ask what is the significance of our definitions of generalization, inclusion, and equivalence, and whether the definition of equivalence corresponds to the usual notion of equivalence. These

questions will become clearer in Section 4.4 where we model several conventional classes of schemas by subsets of the Φ-schemas.

We may note here, however, that the notion of "generalization" is not immediate in conventional schemas, but it goes something like this -- say two schemas (or computer programs) have been written to compute some mathematical function, but the first of these schemas does not compute it for all possible cases as the second one does. Then we would say that the second schema is a generalization of the first. As an example, suppose we want to compute the gamma function, rounded off to, say, ten decimal places. One way of doing it is by computing the factorial function, in which case the program would work correctly for the positive integers. Another way is to use any of the converging series for the gamma function. We would then say that the second program (or schema) is a generalization of the first.

4.3.3 Examples

1. Consider the schemas S_b , S_c of Section 4.2.2. We have

$$S_b \equiv S_c \ .$$

2. Consider the schema $S_d = \langle f_d, \Phi_d, P_d \rangle$ where

$$\Phi_d \text{ is } \forall x \ f_+(x,a_1) = a_1 \leftrightarrow (x = a_0) \ ,$$

$$F_d \text{ is } \text{START } y \leftarrow a_1;$$

$$\underline{\text{while}}\ y \neq a_0 \text{ do } y \leftarrow f_+(y,a_1);$$

$$\text{HALT}(a_0) \ ,$$

and

$$P_d \text{ is } \{a_1, f_+\} \ .$$

Comparing S_d with the schema S_a (of Section 4.2.2) we see that

$$S_a \underset{gen}{\leq} S_d \quad ,$$

but not $S_d \underset{gen}{\leq} S_a$, because the characteristic of a commutative field must be a prime (if it is finite), i.e., if I_d is an interpretation for S_d such that $a_0 = a_1$, or $a_0 = f_+(f_+(f_+(a_1,a_1),a_1),a_1) \neq f_+(a_1,a_1)$ etc., then there is no interpretation I_a for S_a such that $I_a/\{a_1,f_+\}$ is isomorphic to $I_d/\{a_1,f_+\}$. Hence S_d is a strict generalization of S_a (we write $S_a \underset{gen}{<} S_d$). Note that the notion of generalization is not synonymous with usefulness, for it may be argued that S_a is more useful than S_d . The notion of generalization is more akin to the notion of subset in the theory of languages, where any language over an alphabet Σ is a subset of the regular language Σ^* .

4.4 Classes of Schemas

4.4.1 Introduction

We now show how most conventional flowchart schemas can be represented as generalized schemas (Φ-schemas), and demonstrate that many of the well known results regarding the power of classes of schemas apply to Φ-schemas as well. In fact, it even turns out that formalizing a schema as a Φ-schema sometimes reveals some point overlooked when talking about schemas in an informal way. To illustrate, suppose we wish to define conventional schemas with lists, and we introduce the primitives car , cdr , cons , Λ , and atom , and allow their free use in schemas (see also Morris [1972]), then we would find that we cannot

prove the well foundedness of the corresponding generalized schema. The reason is that certain error conditions may be encountered where the computation is not well defined, e.g. in attempting to take the car of Λ of or an atom . This accounts for our careful definition of list schemas in Section 2.1.2. The notation $\mathcal{C}$(n var) , $\mathcal{C}$() , $\mathcal{C}$(=) , $\mathcal{C}$(pds) , $\mathcal{C}$(list) , $\mathcal{C}$(A) , etc., for conventional schemas (described in Section 2.1) will also be used for the corresponding φ-schemas. In fact, we will call a φ-schema corresponding to a conventional schema a conventional φ-schemas.

We first define the notions of generalization, inclusion and equivalence for partially interpreted conventional schemas (in what follows we will consistently use the superscript * for conventional schemas, for interpretations for them, and for classes of conventional schemas). $\Sigma(S^*)$ denotes the set of function and predicate symbols in S^* . We say I is for P (where P is a set of function and predicate symbols) if I specifies at least all the functions and predicates in P . We use I^* for S^*,P to denote (I^* for S^*) and (I^* for P) .

$S_1^* \underset{gen}{\geq} S_2^*$ (let P denote $\Sigma(S_1^*) \cup \Sigma(S_2^*)$): $\forall I_1^*$ for S_1^*,P $\exists I_2^*$ for S_2^*,P $\exists\theta$: $(I_1^*/P) \leftrightarrow (I_2^*/P)$ s.t. either both $Val(S_1^*,I_1^*)$ and $Val(S_2^*,I_2^*)$ are undefined, or else $Val(S_2^*,I_2^*) = \theta(Val(S_1^*,I_1^*))$.

$S_1^* \geq S_2^*$ (let P denote $\Sigma(S_1^*) \cup \Sigma(S_2^*)$:

(i) $\forall I_1^*$ for S_1^*,P $\exists I_2^*$ for S_2^*,P s.t. $\exists\theta$: $(I_1^*/P) \leftrightarrow (I_2^*/P)$, and if $Val(S_2^*,I_2^*)$ is defined then $Val(S_2^*,I_2^*) = \theta(Val(S_1^*,I_1^*))$,

and (ii) $\forall I_2^*$ for S_2^*,P $\exists I_1^*$ for S_1^*,P s.t. $\exists\theta$: $(I_1^*/P) \leftrightarrow (I_2^*/P)$, and if $Val(S_2^*, I_2^*)$ is defined then $Val(S_2^*,I_2^*) = \theta(Val(S_1^*,I_1^*))$.

$\underline{S_1^* \equiv S_2^*}$ if $S_1^* \underset{gen}{\geq} S_2^*$ and $S_2^* \underset{gen}{\geq} S_1^*$; or alternatively, if $S_1^* \geq S_2^*$ and $S_2^* \geq S_1^*$.

We had not defined the notion of generalization for conventional schemas before, but it can be checked that the above definitions for inclusion and equivalence are the same as the earlier definitions for the schemas considered in Chapters 1-3. The earlier definitions, however, do not apply to "arbitrary" partially interpreted conventional schemas.

The translation of conventional schemas to φ-schemas will be performed as follows. In the φ-schemas, symbols used for the base functions and predicates (corresponding to those in the conventional schemas) are distinguished from those used for the interpreted features. Given a conventional schema S^* over the base functions and predicates P , we construct a flowchart F and a formula φ such that the corresponding φ-schema is $S = \langle F,\varphi,P\rangle$. Next, given a class $\mathcal{C}^*$ of conventional schemas, the corresponding class $\mathcal{C}$ of φ-schemas is constructed as follows: if $S^* \in \mathcal{C}^*$, then the corresponding $S = \langle F,\varphi,P\rangle$ is in $\mathcal{C}$, and so are schemas $\langle F,\varphi,P'\rangle$ where $P \subset P'$, but P' may contain some new function and predicate symbols. The reason for this is that if we wish to compare (for inclusion or equivalence) two conventional schemas whose corresponding φ-schemas are $\langle F_1,\varphi_1,P_1\rangle$ and $\langle F_2,\varphi_2,P_2\rangle$, it is possible that $P_1 \neq P_2$; hence we will compare, instead, $\langle F_1,\varphi_1,P_1 \cup P_2\rangle$ with $\langle F_2,\varphi_2,P_1 \cup P_2\rangle$

After we describe the translation of conventional schemas to φ-schemas, we can then go about reproving most of the results regarding conventional schemas in the φ-schema formalism. However, much of this work can be

avoided if the translation process obeys the conditions of the basic translation lemma below. The lemma says that if certain conditions are satisfied then many of the interesting results for conventional schemas carry over to φ-schemas as well.

Let S^* be a conventional schema, and let its statements be $s_0, s_1, \ldots, s_k$. A statement can be of "type" -- start, halt, loop, assignment, or test. The flowchart F of the corresponding φ-schema $S = \langle F, \Phi, P \rangle$ will have one statement corresponding to each statement in S^*, and the types match, and $P = \Sigma(S^*)$. For convenience, we will call the statements in F by the same names as those in S^*, i.e., $s_0, s_1, \ldots, s_k$.

The conditions for the basic translation lemma are the following (we use the notation $I_1 \leftrightarrow I_2$ to denote "I_1 and I_2 are isomorphic"):

0. S is well founded.

1. (For individual schemas) Let $P_+ \supset P = \Sigma(S^*)$.

(a) $\forall I_+$ for P_+ if $\exists I$ for S s.t. $(I/P) \leftrightarrow (I_+/P)$ then $\exists I_1$ for S, P_+ s.t. $(I_1/P_+) \leftrightarrow (I_+/P_+)$.

(b) $\forall I_+^*$ for P_+ if $\exists I^*$ for S^* s.t. $(I^*, P) \leftrightarrow (I_+^*/P)$ then $\exists I_1^*$ for S^*, P_+ s.t. $(I_1^*/P_+) \leftrightarrow (I_+^*/P_+)$.

2. (For the translation process)

(a) $\forall I$ for S $\exists I^*$ for S^* s.t. $\exists \Theta$: $(I^*/P) \leftrightarrow (I/P)$ and $\mathrm{Path}(S,I) = \mathrm{Path}(S^*,I^*)$, and $\mathrm{Val}(S,I) = \Theta(\mathrm{Val}(S^*,I^*))$ if both are defined.

(b) $\forall I^*$ for S^* $\exists I$ for S s.t. $\exists \Theta$: $(I^*/P) \leftrightarrow (I/P)$ and $\mathrm{Path}(S,I) = \mathrm{Path}(S^*,I^*)$, and $\mathrm{Val}(S,I) = \Theta(\mathrm{Val}(S^*,I^*))$ if both are defined.

3. (For classes of conventional schemas) Interpolation lemma.
$\forall \mathcal{C}_1^*, \mathcal{C}_2^*, S_1^*, S_2^*$, if $S_1^* \in \mathcal{C}_1^*$, $S_2^* \in \mathcal{C}_2^*$, $S_1^* \equiv S_2^*$ then
$\exists S_3^* \in \mathcal{C}_2^*$ s.t. $S_1^* \equiv S_3^*$, $\Sigma(S_1^*) = \Sigma(S_3^*)$.

It is easy to see that for uninterpreted conventional schemas, 2(a) follows from 2(b) owing to the well foundedness of S . To see that this is indeed the case, let I be any interpretation for S . Then, as S^* is uninterpreted, there is an I^* for S^* such that I^*/P is isomorphic to I/P , i.e., there is an isomorphism $\theta : (I^*/P) \leftrightarrow (I/P)$. Now, from part 2(b), there is an interpretation I_1 for S such that $\theta_1 : (I^*/P) \leftrightarrow (I_1/P)$, and $Path(S,I_1) = Path(S^*,I^*)$, and $Val(S,I_1) = \theta_1(Val(S^*,I^*))$. But from the well foundedness of S , as $\theta_1 \circ \theta^{-1} : (I/P) \leftrightarrow (I_1/P)$, we have $Path(S,I) = Path(S,I_1)$, and $Val(S,I_1) = \theta_1 \circ \theta^{-1}(Val(S,I))$, from which the desired result follows, i.e., $Path(S,I) = Path(S^*,I^*)$, and $Val(S,I) = \theta \circ \theta_1^{-1} \circ \theta_1(Val(S^*,I^*)) = \theta(Val(S^*,I^*))$.

If we can prove the above condition to hold in the translation process, then the following consequences apply.

For individual schemas

(1) S halts if and only if S^* halts, and in general, S halts on I if and only if S^* halts on (I/P) .

(2) S diverges if and only if S^* diverges, and in general, S diverges on I if and only if S^* diverges on (I/P) .

(3) If $\Sigma(S_1^*) = \Sigma(S_2^*)$ then $S_1 \underset{gen}{\leq} S_2$ iff $S_1^* \underset{gen}{\leq} S_2^*$.

(4) If $\Sigma(S_1^*) = \Sigma(S_2^*)$ then $S_1 \leq S_2$ iff $S_1^* \leq S_2^*$.

(5) If $\Sigma(S_1^*) = \Sigma(S_2^*)$ then $S_1 \equiv S_2$ iff $S_1^* \equiv S_2^*$.

(6) S is free iff S^* is free.

For classes of schemas

(7) $\mathcal{C}_1 \leq \mathcal{C}_2$ iff $\mathcal{C}_1^* \leq \mathcal{C}_2^*$.

(8) $\mathcal{C}_1 \equiv \mathcal{C}_2$ iff $\mathcal{C}_1^* \equiv \mathcal{C}_2^*$

(9) The halting problem (respectively divergence, equivalence, inclusion problem) is solvable for $\mathcal{C}$ if and only if it is solvable for $\mathcal{C}^*$.

For a proof, see Section 4.6.

In our translations from conventional schemas to φ-schemas we show that the basic translation lemma applies by proving part 2(b) above, by induction on the number of steps in the computation. This is done as follows. Given an interpretation I^* for S^* , we construct an interpretation I for S such that (I^*/P) is isomorphic to (I/P) , and we define a function $\delta: M \to Dom(I)$ where M is the set of possible configurations of the data space (memory) of S^* . Then we show that at each step in the computations of S^* and S , the configuration of the data space in S^* and the value of the variable y of S are related by the function δ .

4.4.2 Flowchart Schemas

4.4.2. One-Variable Schemas

For Ianov schemas, and general one-variable flowchart schemas with equality tests (but without boolean variables), the translation to φ-schemas is trivial. Given a one-variable schema S^* the corresponding φ-schema is $S = \langle F, true, P \rangle$, where F is identical to the flowchart of S^* .

Proving that the basic translation lemma applies in this case is also trivial. Since the set P of functions and predicates in S^* is the same as the set of functions and predicates of S , S is well

founded. Now, given an interpretation I^* for S^*, choose I to be the same as I^*, then the set of memory values of S^* is just $Dom(I^*)$, and by choosing δ to be the identity function we see that the condition of the basic translation lemma is satisfied.

4.4.2.2 n-variable Schemas

Given an n-variable flowchart schema S^* with variables $y_1, y_2, \ldots, y_n$, no boolean variables, and predicates and functions P, to construct $S = \langle F, \varphi, P \rangle$, we add $(n+1)$ new functions: $comb, v_1, v_2, \ldots, v_n$. The formula φ is:

$$\begin{aligned} \forall x_1 \forall x_2 \ldots \forall x_n \quad & v_1(comb(x_1, x_2, \ldots, x_n)) = x_1 \\ & \wedge\ v_2(comb(x_1, x_2, \ldots, x_n)) = x_2 \\ & \wedge\ \ldots \\ & \wedge\ v_n(comb(x_1, x_2, \ldots, x_n)) = x_n \ . \end{aligned}$$

To construct the flowchart F we first define the translation $T(\tau(y_1, \ldots, y_n))$ of a term $\tau(y_1, \ldots, y_n)$ which uses the functions of P and the variables $y_1, \ldots, y_n$ (any or all of them may be missing). The translated term uses only the functions from $P \cup \{v_1, \ldots, v_n\}$ and the variable y. The translation may be defined as follows:

(a) $T(\tau()) = \tau()$,

(b) $T(y_i) = v_i(y)$,

(c) $T(f(\tau_1, \ldots, \tau_k)) = f(T(\tau_1), \ldots, T(\tau_k))$, where f is a k-ary function letter.

We can now define the statements of the flowchart F by setting up a correspondence from statements of the schema S^*.

Statement of S^*	Statement of F
START $\langle y_1,\ldots,y_n\rangle \leftarrow \langle \tau_1(),\ldots,\tau_n()\rangle$	START $y \leftarrow \text{comb}(\tau_1(),\ldots,\tau_n())$
HALT(τ)	HALT($T(\tau)$)
LOOP	LOOP
$\tau_1 = \tau_2$	$T(\tau_1) = T(\tau_2)$
$p(\tau_1,\ldots,\tau_k)$	$p(T(\tau_1),\ldots,T(\tau_k))$
$\langle y_1,\ldots,y_n\rangle \leftarrow \langle \tau_1,\ldots,\tau_n\rangle$	$y \leftarrow \text{comb}(T(\tau_1),\ldots,T(\tau_n))$

We can prove the well foundedness of S, and the basic translation lemma simultaneously by induction on the number of steps of the computation.

Given an interpretation I^* for S^* we can get an interpretation I for S (such that $I \models \varphi$) as follows: the domain of I, $\text{Dom}(I)$ is defined to be the closure of the following:

(a) $\text{Dom}(I^*) \subset \text{Dom}(I)$

(b) if $e_1,e_2,\ldots,e_n \in \text{Dom}(I)$ then the vector $\langle e_1,\ldots,e_n\rangle \in \text{Dom}(I)$ (without loss of generality we may assume that vectors like this are not already present in $\text{Dom}(I^*)$).

The functions and predicates of P are defined as follows: if q is a k-ary function or predicate, $q \in P$, then $q(e_1,\ldots,e_n)$ in I is defined to equal the value of $q(e_1,\ldots,e_n)$ in I^* if $e_1,\ldots,e_n$ are all elements of $\text{Dom}(I^*)$, otherwise it is arbitrary. The function comb is defined as follows:

$$\text{comb}(e_1,e_2,\ldots,e_n) = \langle e_1,\ldots,e_n\rangle .$$

The functions $v_1,\ldots,v_n$ are defined as follows: if $e \in Dom(I^*)$ then $v_i(e)$ is arbitrary, otherwise e is a vector of elements in $Dom(I)$, $e = \langle e_1,\ldots,e_n\rangle$, and $v_i(e) = e_i$.

Now, the data space of S^* at any instant is a set of values $\{y_1 = e_1, y_2 = e_2, \ldots, y_n = e_n\}$ where $e_1,\ldots,e_n$ are elements of $Dom(I^*)$. We define the function δ mapping this data space into the element $\langle e_1,\ldots,e_n\rangle$ of $Dom(I)$. Also, it is clear that I^*/P and I/P are isomorphic.

Now the induction hypothesis after i steps in the computations of S and S^* (under I and I^* respectively) is that the paths up to that point are the same, and that $v = \delta(m)$ where m is the data space of S^* after i steps, and v is the value of the variable y of S. The initial step and the induction step of the proof are easy to check.

We remark here that there are other possible translations of n-variable schemas to φ-schemas that yield relatively more natural interpretations I corresponding to I^*. We give an example below. Here, we introduce the same functions as before, i.e., $comb, v_1,\ldots,v_n$, but also a new predicate: isdata. Let $f_1, f_2, \ldots$ be the functions of P (including zero-ary functions), and let r be the largest rank of all these; then φ is

$$\begin{aligned}
&\forall x_1 \ldots \forall x_r\ (isdata(x_1) \wedge \ldots \wedge isdata(x_r)) \rightarrow \\
&\qquad isdata(f_1(x_1,x_2,\ldots)) \\
&\qquad \wedge\ isdata(f_2(x_1,x_2,\ldots)) \\
&\qquad \wedge\ \ \vdots \\
&\wedge\ \forall x_1 \ldots \forall x_n\ (isdata(x_1) \wedge \ldots \wedge isdata(x_r)) \rightarrow \\
&\qquad v_1(comb(x_1,\ldots,x_n)) = x_1 \\
&\qquad \wedge\ v_2(comb(x_1,\ldots,x_n)) = x_2 \\
&\qquad \wedge\ \ \vdots
\end{aligned}$$

and the flowchart F is the same as in the earlier construction. In this construction, the domain of the interpretation I need not contain vectors whose elements are also vectors. However, it should be noted that if these two translations yield schemas S_1 and S_2 corresponding to a conventional schema S^*, then $S_1 \equiv S_2$.

4.4.3 Flowchart Schemas with Markers and Boolean Variables

4.4.3.1 Markers

Give a flowchart schema S^* with n variables $y_1,\ldots,y_n$, m marker variables $z_1,\ldots,z_m$, and p marker constants $M_1,\ldots,M_p$, and predicates and functions P, to construct $S = \langle F,\varphi,P\rangle$ we add $(p+m+n+1)$ new functions: $comb, v_1,\ldots,v_n, w_1,\ldots,w_m, M_1,\ldots,M_p$. The formula φ is:

$$M_1 \neq M_2 \wedge M_1 \neq M_3 \wedge \ldots \wedge M_{p-1} \neq M_p$$

$$\wedge \forall y_1 \forall y_2 \ldots \forall y_n \forall z_1 \ldots \forall z_m \quad v_1(comb(y_1,\ldots,z_m)) = y_1$$

$$\wedge \quad \ldots$$

$$\wedge\ v_n(comb(y_1,\ldots,z_m)) = y_n$$

$$\wedge\ w_1(comb(y_1,\ldots,z_m)) = z_1$$

$$\wedge \quad \ldots$$

$$\wedge\ w_m(comb(y_1,\ldots,z_m)) = z_m \quad .$$

The flowchart F is obtained on lines very similar to that described in Section 4.4.2.2. The addition is that a test $(z_i = M_j)$ is translated to a test $(w_i(y) = M_j)$ -- note that M_j in the test $(z_i = M_j)$ corresponds to a marker, whereas in the test $(w_i(y) = M_j)$, the M_j is a zero-ary function.

Well foundedness of S and the basic translation lemma can be proved as before by constructing the function δ and using the additional induction hypothesis that at any point in the computation the value of each z_j, $1 \leq j \leq m$, is M_1 or M_2 or ... or M_p.

Flowchart schemas with boolean variables can be treated as marker-schemas where the markers can have one of two values called "true" and "false".

4.4.3.2 Generic Variables

A generic variable in a conventional schema is an untyped variable whose value can be either a data element or a marker -- in other words, the "type" is assigned at run-time rather than at compile time. Schemas with generic variables differ from other schemas in that there can be an "unexpected" error condition of type mismatch. Under such conditions the schema is assumed to loop.

Given a flowchart schema S^* with n generic variables $y_1,\ldots,y_n$, p marker constants $M_1,\ldots,M_p$, and function symbols $f_1,\ldots,f_m$ with rank $r_1,\ldots,r_m$ respectively (some of the r's may be zero), let r denote $\max(r_1,\ldots,r_m)$. Now, the corresponding φ-schema $S = \langle F,\varphi,P\rangle$ is given as follows. We introduce m+p+2 new functions: $M_1,\ldots,M_p,\text{ism},\text{comb},v_1,\ldots,v_n$. The formula φ is:

$$M_1 \neq M_2 \wedge M_1 \neq M_3 \wedge \ldots \wedge M_{p-1} \neq M_p$$

$$\wedge\ \mathrm{ism}(M_1) \wedge \mathrm{ism}(M_2) \wedge \ldots \wedge \mathrm{ism}(M_p)$$

$$\wedge\ \forall x_1 \forall x_2 \ldots \forall x_r (\neg\ \mathrm{ism}(x_1) \wedge \ldots \wedge \neg\ \mathrm{ism}(x_r)) \rightarrow$$

$$\neg\ \mathrm{ism}(f_1(x_1,\ldots,x_{r_1}))$$

$$\wedge\ \ldots$$

$$\wedge \neg\ \mathrm{ism}(f_m(x_1,\ldots,x_{r_m}))$$

$$\wedge\ \forall x_1 \forall x_2 \ldots \forall x_r\ v_1(\mathrm{comb}(x_1,\ldots,x_n)) = x_1$$

$$\wedge\ \ldots$$

$$\wedge\ v_n(\mathrm{comb}(x_1,\ldots,x_n)) = x_n \quad .$$

The flowchart F can be defined by setting up a correspondence between statements of S^* and statements of F. Without loss of generality we assume that no statement of S^* applies a function or predicate to a marker constant (for it can be replaced by the loop statement). We will use the function T defined in Section 4.4.2.2, extended to include markers by letting $T(M_i) = M_i$. If $\tau_1,\ldots,\tau_k$ are terms we use $Y(\tau_1,\ldots,\tau_k)$ to denote the set of variables y_i appearing in $\tau_1,\ldots,\tau_k$, and if $Y = \{y_{k_1},\ldots,y_{i_s}\}$ is any set of variables, we use $\mathrm{ism}(Y)$ as an abbreviation for $(\mathrm{ism}(y_{i_1}) \vee \mathrm{ism}(y_{i_2}) \vee \ldots \vee \mathrm{ism}(y_{i_s}))$.

Statement of S^*	Statement of F
START $\langle y_1,\ldots,y_n\rangle \leftarrow \langle\tau_1(),\ldots,\tau_n()\rangle$	START $y \leftarrow \text{comb}(\tau_1(),\ldots,\tau_n())$
HALT(τ)	if ism(Y(τ)) then LOOP else HALT(τ)
LOOP	LOOP
$y_i = M_j$	$y_i = M_j$
if $p(\tau_1,\ldots,\tau_k)$ then goto L_1 else goto L_2	if ism(Y($\tau_1,\ldots,\tau_k$)) then LOOP else if $p(T(\tau_1),\ldots,T(\tau_k))$ then goto L_1 else goto L_2
$\langle y_1,\ldots,y_n\rangle \leftarrow \langle\tau_1,\ldots,\tau_n\rangle$	if ism(Y($\tau_{i_1},\ldots,\tau_{i_k}$)) then LOOP else $y \leftarrow \text{comb}(T(\tau_1),\ldots,T(\tau_n))$

where $\tau_{i_1},\ldots,\tau_{i_k}$ are the terms in $\tau_1,\ldots,\tau_n$ that contain at least one function symbol. It can be shown by induction that the φ-schema S is well founded. However, the translation does not satisfy the basic translation lemma to the letter because extra tests are introduced. This, however, does not violate the spirit of the lemma inasmuch as all properties except freedom are considered.

4.4.4 Counters, Stacks, Arrays, and Other Features

In this section a conventional flowchart schema is assumed to have a finite number of discrete elements: variables, counters, stacks, arrays, queues, lists, etc. In the corresponding φ-schema, the mechanism of the functions comb , $v_1,\ldots,v_n$ is used to assemble and to extract the various components as in the earlier sections, and the

corresponding axioms will not be repeated. Similarly, the assignment to variables, and predicate tests, as well as halt and loop statements are handled as before. In this section we will concentrate only on the translation of these special features into φ-schemas.

4.4.4.1 Counters

The operations allowed on counters are setting a counter to zero, testing a counter for zero, and incrementing and decrementing a counter (decrementing a counter whose value is zero leaves it unchanged).

To translate a counter schema into a φ-schema we introduce three new functions: a zero-ary function zero, and two unary functions plusone and minusone. The axioms are:

$$\forall x(\text{plusone}(x) \neq x)$$

$$\forall x\ \text{minusone}(\text{plusone}(x)) = x$$

$$\text{minusone}(\text{zero}) = \text{zero}\ .$$

Note that the axiom " $\forall x\ \text{plusone}(x) \neq \text{zero}$ " follows from these.

We see that we can define some new features within the framework of φ-schemas very easily:

(i) counters that take positive and negative values

(ii) testing two counters for equality

(iii) comparison of two counters

(iv) addition and multiplication of counters

(v) "counters" that take on rational values

(vi) schemas that can output counter values. On the other hand, inputing an arbitrary counter value is restricted, owing to the first order notions of φ-schemas.

4.4.4.2 Arrays

One dimensional semi-infinite arrays without booleans can be "described" by using functions con and ass (which stand for "contents", and "assignment" respectively). Con(c,A) represents the contents of array A at location c, and ass(x,c,A) represents the array obtained by assigning the value of array A at location c to be x.

$$\forall x \forall c \forall c' \forall a \quad con(c,ass(x,c,a)) = x$$

$$\wedge\ c' \neq c \rightarrow con(c',ass(x,c,a)) = con(c',a) \quad .$$

The value of A[c] is translated to con(c,A), and an assignment A[c] ← y is translated to ass(y,c,A).

The start statement is used to initialize all the locations of an array to some constant term $\tau()$. For this, we introduce a zero-ary function "init" in the Φ-schema which represents an array with all its locations having value f, by the axiom

$$\forall c \ con(c,init) = \tau() \quad .$$

In like manner we can define arrays whose locations take data, boolean and marker values, multidimension arrays, arrays that are infinite in both directions, and an interesting feature: arrays that are referenced by terms.

4.4.4.3 Pushdown Stacks

One-track Stacks

A conventional schema with a one-track pushdown stack can push data values on top of the stack, pop them, look at the top element of the stack, and test the stack to see if it is empty. Statements allowed are:

(1) $s \leftarrow push(s,y)$

(2) if $s = \Lambda$ then goto L

else begin $y \leftarrow top(s)$; $s \leftarrow pop(s)$ end .

We introduce the functions: top , pop , push , and Λ . The axioms are self-explanatory:

$$\forall s \forall x \quad push(s,x) \neq \Lambda$$
$$\wedge\ top(push(s,x)) = x$$
$$\wedge\ pop(push(s,x)) = s \quad .$$

The resulting φ-schema we get is well founded. However, if in the original conventional schema we allowed arbitrary use of push , top , and pop , e.g., if statements allowed were

(1) $s \leftarrow push(s,y)$

(2) if $s = \Lambda$ then goto L_1 else goto L_2

(3) $y \leftarrow top(s)$

(4) $s \leftarrow pop(s)$

then the resulting φ-schema may not be well founded. And with good reason. The operation of the original schema may not be well defined for all cases, e.g., what happens when an empty stack is popped? As an added axiom we can specify

$$pop(\Lambda) = \Lambda$$

but the φ-schema may still not be well founded. The value of $top(\Lambda)$ is undefined. To overcome this, we may specify that there are an infinite number of data elements " a " (a zero-ary function), at the bottom of an "empty" stack; we then have the axiom

$$top(\Lambda) = a$$

and the resulting schema is finally well founded.

Two-track Stacks

A stack with two tracks has one track for data values, and one for markers (booleans can be represented as markers). We could allow markers and data values to be mixed in a single track, but we again have the ad-hoc condition that the schema loops in case of type-checking error. This is the notion of a stack introduced in Section 2.1.2. The statements allowed are:

(1) $s \leftarrow push(s,y,z)$

(2) if $s = \Lambda$ then goto L

else begin $y \leftarrow top_1(s)$; $z \leftarrow top_2(s)$; $s \leftarrow pop(s)$ end .

The axioms are:

$$\begin{aligned} \forall x \forall s \forall m \quad & push(s,x,m) \neq \Lambda \\ & \wedge\ top_1(push(s,x,m)) = x \\ & \wedge\ top_2(push(s,x,m)) = m \\ & \wedge\ pop(push(s,x,m)) = s \quad . \end{aligned}$$

4.4.4.4 Queues

A schema with a one-track queue can insert a value at one end of the queue, can test to see if the queue is empty, and if it is not the schema can look at, or delete a value at the other end. The axioms:

$$\begin{aligned} \forall x \forall q \quad & add(q,x) \neq \Lambda \\ & \wedge\ first(add(\Lambda,x)) = x \\ & \wedge\ remove(add(\Lambda,x)) = \Lambda \\ & \wedge\ (q \neq \Lambda) \rightarrow first(add(q,x)) = first(q) \\ & \wedge\ (q \neq \Lambda) \rightarrow remove(add(q,x)) = add(remove(q,x)) \quad . \end{aligned}$$

A two-track queue is a queue that has two tracks, one for data values and one for markers (see Section 2.1.2). The axioms are:

$$\begin{aligned}
&\forall x \forall q \forall m \quad add(q,x,m) \neq \Lambda \\
&\quad \wedge\ first_1(add(\Lambda,x,m)) = x \\
&\quad \wedge\ first_2(add(\Lambda,x,m)) = m \\
&\quad \wedge\ remove(add(\Lambda,x,m)) = \Lambda \\
&\quad \wedge\ (q \neq \Lambda) \rightarrow first_1(add(q,x,m)) = first_1(q) \\
&\quad \wedge\ (q \neq \Lambda) \rightarrow first_2(add(q,x,m)) = first_2(q) \\
&\quad \wedge\ (q \neq \Lambda) \rightarrow remove(add(q,x,m)) = add(remove(q,x,m)) \quad .
\end{aligned}$$

4.4.4.5 Lists

Axioms for lists are very similar to the axioms for pushdown stacks. The schemas differ mainly in the type of statements allowed (see Section 2.1.2), for if stack schemas allowed the construction of a stack of stacks, and a stack of stack of stacks, etc., we would have a list structure.

Let $f_1,\ldots,f_m$ denote the function symbols of the schema, let their ranks be $r_1,\ldots,r_m$, and let $r = \max(r_1,\ldots,r_m)$. We have

$$\begin{aligned}
&atom(\Lambda) \\
&\wedge\ \forall x_1 \ldots \forall x_r \quad atom(f_1(x_1,\ldots,x_{r_1})) \wedge (f_1(x_1,\ldots,x_{r_1}) \neq \Lambda) \\
&\qquad\qquad \wedge \quad \ldots \\
&\qquad\qquad \wedge\ atom(f_m(x_1,\ldots,x_{r_m})) \wedge (f_m(x_1,\ldots,x_{r_m}) \neq \Lambda) \\
&\wedge\ \forall x_1 \forall x_2 \neg\ atom(cons(x_1,x_2)) \\
&\wedge\ \forall x_1 \forall x_2 \quad car(cons(x_1,x_2)) = x_1 \\
&\qquad\qquad \wedge\ cdr(cons(x_1,x_2)) = x_2 \quad .
\end{aligned}$$

4.5 Properties of Generalized Schemas

4.5.1 Interpreted Schemas, Herbrand Schemas, and Oracle Schemas

When we say that a conventional schema is uninterpreted, we mean that any interpretation over its base functions is relevant for the schema. We say it is uninterpreted even though its structural features are interpreted, e.g., the operation of pushing a value into a stack, or of incrementing a counter, is well defined. We would like to make this notion somewhat more formal, and apply it to our generalized schemas.

Definition. A well founded schema $S = \langle F,\Phi,P\rangle$ is said to be uninterpreted if for every interpretation I for P there is an interpretation I' for S whose subinterpretation over P is isomorphic to I, i.e.,

$$\forall I \text{ for } P,\ \exists I' \text{ for } S, \text{ i.e., } I' \models \Phi, \text{ such that}$$
$$\exists \text{ an isomorphism } \theta: (I/P) \leftrightarrow (I'/P) .$$

Note: we use (I/P) above instead of I because there may be some elements in $Dom(I)$ that are not reachable, i.e., not expressible in terms of the functions of P (and, of course, there may be some functions and predicates defined in I that are not in P).

As an example, let Φ_a denote

$$\forall x\ f(g(x)) = g(f(x)) = x$$

and let F_a denote

START y ← f(a);
<u>while</u> p(y) <u>do</u> y ← f(y);
HALT(g(y)) ,

then $S_a = \langle F_a,\Phi_a,\{a,f,p\}\rangle$ is uninterpreted, but $S_a' = \langle F_a,\Phi_a,\{a,f,g,p\}\rangle$

is not. Note that both S_a and S'_a are well founded, but $\langle F_a, \varphi_a, \{a,g,p\}\rangle$ is not.

For another example, let φ_b be the same as φ_a, and F_b be

START $y \leftarrow a$;

<u>while</u> $p(y)$ <u>do</u> $y \leftarrow f(y)$;

HALT$(g(y))$.

Now $P_b = \{a,f,g,p\}$ is the minimal set for which $S_b = \langle F_b, \varphi_b, P_b\rangle$ is well founded, and S_b is not uninterpreted.

We should note that all the conventional φ-schemas (i.e., φ-schemas corresponding to $\mathcal{C}$(marker,pds,q,list,A)) are uninterpreted schemas.

If H is the Herbrand interpretation corresponding to an interpretation I (see definition in Section 2.1.7), we write $I \overset{h}{\rightarrow} H$.

<u>Definition</u>. A well founded schema $S = \langle F,\varphi,P\rangle$ is called a <u>semi-Herbrand schema</u> if

(a) $\forall I$ for S , $\exists H$ for S , such that $(I/P) \overset{h}{\rightarrow} (H/P)$, and

(b) $\forall H$ for S , $\forall I_1$ such that $(I_1/P) \overset{h}{\rightarrow} (H/P)$, $\exists I$ for S , such that $(I_1/P) = (I/P)$.

Note that the definition of a semi-Herbrand schema depends only on φ and P , and not really on F . Saying that a schema S is semi-Herbrand simply means that for every interpretation for S the corresponding Herbrand interpretation is allowed for S , and that for every Herbrand interpretation for S all corresponding interpretations are also allowed for S . Any uninterpreted schema is semi-Herbrand, as is any schema in which φ is equality-free.

Definition. A semi-Herbrand schema $S = \langle F,\varphi,P\rangle$ is said to be a Herbrand schema if

$\forall I,H$ for S, if $(I/P) \overset{h}{\rightarrow} (H/P)$ then $Path(S,I) = Path(S,H)$, and $Val(S,I)$ corresponds to $Val(S,H)$.

Note that $Val(S,I)$ and $Val(S,H)$ correspond in the obvious sense, i.e., $Val(S,I)$ is the value in I of the term $Val(S,H)$ of functions of P.

By this definition it is clear that all the conventional φ-schemas without equality tests (in the flowcharts) are Herbrand schemas (see also Theorem 2.3, Section 2.1.7). This is not true, however for the φ-schemas in general, for consider the schema $S_c = \langle F_c,\varphi_c,P_c\rangle$ where

φ_c is $\forall x\forall y\ p(x,y) \leftrightarrow (x = y)$

F_c is START $y \leftarrow a_1$; if $p(y,a_2)$ then HALT(y) else LOOP

and

P_c is $\{a_1,a_2,p\}$.

S_c is not a Herbrand schema because for the interpretation I where $a_1 = a_2$ and $p(a_1,a_2)$ is true, there is no corresponding Herbrand interpretation for S_c. Further, $S_c' = \langle F_c,\varphi_c,\{a_1,a_2\}\rangle$ is also non-Herbrand because the interpretation H corresponding to I has a_1 = the term "a_1", a_2 = the term "a_2", and $p(a_1,a_2)$ = false, but the paths for I and for H are not the same. So, we see that we can obtain the effect of equality tests without actually using them in the flowchart.

We should mention that the notions of interpreted schemas and Herbrand schemas are independent. Both S_c and S_c' above are non-Herbrand, but S_c is interpreted, whereas S_c' is uninterpreted. Also, consider φ_d and F_d below:

Φ_d is $\forall x\ p(x) \leftrightarrow \neg\, p(f(x))$

and

F_d is START $y \leftarrow a$; if $p(y)$ then HALT(y) else LOOP .

Both $S_d = \langle F_d, \Phi_d, \{a,f,p\}\rangle$, and $S_d' = \langle F_d, \Phi_d, \{a,p\}\rangle$ are Herbrand schemas, but S_d is interpreted, whereas S_d' is uninterpreted.

Given a class $\mathcal{I}$ of interpretations, a schema S is said to halt on $\mathcal{I}$ if S halts on every interpretation I for S , where $I \in \mathcal{I}$; and similarly for divergence and freedom. And we say that $S_1 \underset{\text{gen}}{\leq} S_2$ on $\mathcal{I}_1, \mathcal{I}_2$ if $\forall I_1$ for S_1 , $I_1 \in \mathcal{I}_1$, $\exists I_2$ for S_2 , $I_2 \in \mathcal{I}_2$, and $\exists$ an isomorphism θ: $(I_1/P) \leftrightarrow (I_2/P)$ such that either both schemas diverge, or $\mathrm{Val}(S_2,I_2) = \theta(\mathrm{Val}(S_1,I_1))$ -- compare with the definition of $S_1 \underset{\text{gen}}{\leq} S_2$. And similarly for inclusion and equivalence.

Given schemas $S_1 = \langle F_1, \Phi_1, P\rangle$ and $S_2 = \langle F_2, \Phi_2, P\rangle$, let $\mathcal{K}_1$ be the class of interpretations H for S_1 such that (H/P) is a Herbrand interpretation; and similarly for $\mathcal{K}_2$, then:

Theorem 4.1 (Fundamental theorem of Herbrand schemas)

For Herbrand schemas S_1 , S_2

(a) S_1 halts if and only if it halts on $\mathcal{K}_1$,

(b) S_1 diverges if and only if it diverges on $\mathcal{K}_1$,

(c) $S_1 \equiv S_2$ if and only if $S_1 \equiv S_2$ on $\mathcal{K}_1, \mathcal{K}_2$,

(d) $S_1 \leq S_2$ if and only if $S_1 \leq S_2$ on $\mathcal{K}_1, \mathcal{K}_2$,

(e) $S_1 \underset{\text{gen}}{\leq} S_2$ if and only if $S_1 \underset{\text{gen}}{\leq} S_2$ on $\mathcal{K}_1, \mathcal{K}_2$, and

(f) S_1 is free if and only if S_1 is free on $\mathcal{K}_1$.

For the proof, see Section 4.6. This theorem is an extended and relatively more formal version of Theorem 2.1.2 (in which the class of Herbrand schemas was comparatively restricted).

There is another property about conventional schemas that we would like to capture. It is that in a single step a conventional schema can do only a "small" amount of work, i.e., it can execute an assignment statement or make an atomic test. We can generalize the notion of a schema to what may be called a "logic-theory machine". A logic-theory machine is like an ordinary schema except that it can also make quantified tests, and in general, a test can be any well formed formula (an even more "powerful" machine would be one that can also build up formulas as strings, or trees). A test that effectively looks at an infinite number of values may be called an oracle test, and a "schema" that can make such tests may be called an oracle schema.

Definition. We say that a formula Ψ is over a set P of function and predicate symbols if it uses no function or predicate symbols other than those in P.

Definition. Given a well-founded schema $S = \langle F,\varphi,P\rangle$, we say that S is a non-oracle schema if

(a) for every path in F from the start statement to a test, there exists a quantifier free formula $\Psi()$ over P such that for all interpretations (for S) that follow this path, the outcome (true or false) of the test equals the value of $\Psi()$ for the interpretation, and

(b) for every path in F from the start statement to a halt statement, there is a quantifier free formula $\Psi(x)$ over P such that all interpretations (for S) that follow the path, for all elements x in the interpretation, the output of the halt statement is x if and only if $\Psi(x)$ is true.

Lemma 4.2

Every well-founded schema is a non-oracle schema.

This property of schemas (proved in Section 4.6) is an important one, and is used in the proof of the theorem of maximal schemas (Theorem 4.3).

4.5.2 The Fundamental Theorem of Maximal Schemas

Constable and Gries [1972] suggested that the class of (conventional) schemas with arrays, $\mathcal{C}(A)$, are a maximal class of (uninterpreted) schemas. Chandra and Manna [1972] showed that for a "reasonable" definition of uninterpreted schemas, arrays, by themselves, are not adequate, and that equality tests too are required -- and that the class $\mathcal{C}(A,=)$ is strictly more powerful than $\mathcal{C}(A)$. We show here that the class $\mathcal{C}(A,=)$ is indeed maximal in our generalized schema formalism.

Theorem 4.3 (Theorem of maximal schemas)

The class $\mathcal{C}$ of uninterpreted schemas is equivalent to the class $\mathcal{C}(A,=)$ of generalized schemas corresponding to the conventional schemas with arrays and equality tests; and, in fact, a schema in $\mathcal{C}$ can be effectively translated into an equivalent schema in $\mathcal{C}(A,=)$.

For the proof of this theorem, see Section 4.6.

Intuitively it does seem that for conventional schemas, the class $\mathcal{C}(A)$ is indeed "maximal" in some sense. Chandra and Manna [1972] conjectured that $\mathcal{C}(A)$ may be maximal for Herbrand schemas. We show that this is indeed the case for our generalized schema formalism.

Theorem 4.4 (Theorem of maximal Herbrand schemas)

The class $\mathcal{C}$ of uninterpreted Herbrand schemas is equivalent to the class $\mathcal{C}(A)$ of generalized schemas corresponding to the conventional schemas with arrays; and, in fact, a schema in $\mathcal{C}$ can be effectively translated into an equivalent schema in $\mathcal{C}(A)$.

For the proof of this theorem, see Section 4.6.

4.5.3 Decision Problems

We consider the following decision problems for the class of Φ-schemas.

1. The halting problem -- given a Φ-schema S, to decide if it halts for every interpretation for S.
2. The divergence problem -- given a Φ-schema S, to decide if it diverges for every interpretation for S.
3. The equivalence problem -- given two compatible Φ-schemas S_1 and S_2, to decide if they are equivalent. We also consider the generalization problem (to decide if $S_1 \underset{gen}{\leq} S_2$) and the inclusion problem (to decide if $S_1 \leq S_2$).

4.5.3.1 The Halting Problem

Theorem 4.5

The halting problem for Φ-schemas is not solvable, but it is partially solvable.

The unsolvability of the halting problem for Φ-schemas can be shown in many ways (e.g., by using the unsolvability of the halting problem for several classes of conventional schemas), but perhaps the simplest is the following. Consider the class of schemas, of the form $\langle F,\Phi,P\rangle$ where Φ and P are arbitrary, and F is:

START y ← a; LOOP .

Then a schema in the class halts if and only if Φ is unsatisfiable -- which is a well known unsolvable problem.

The proof of the partial solvability of the halting problem is also quite easy, but we defer it to Section 4.6.

4.5.3.2 The Divergence Problem

The complement of the divergence problem is called the non-divergence problem, i.e., given a schema, to decide if it halts for any (relevant) interpretation.

Theorem 4.6

Both the divergence problem and the non-divergence problem for schemas are not partially solvable.

The divergence problem is not partially solvable because the divergence problem for one-variable schemas with equality is not partially solvable (see Chapter 3). The non-divergence problem is not partially solvable because the schema $\langle F,\varphi,\{a\}\rangle$ where F is

START y ← a; HALT(y)

halts for some interpretation if and only if φ is satisfiable -- a problem that is not partially solvable.

It is interesting to note that while the non-divergence problem is partially solvable for all conventional schemas (e.g., those of Section 4.4), it is not partially solvable for φ-schemas. One should ask what it is about φ-schemas that causes this difference. The next theorem attempts to answer this question.

Lemma 4.7. The non-divergence problem for uninterpreted schemas is partially solvable.

This follows directly from the fundamental theorem of maximal schemas and the fact that the divergence problem for the class of conventional array schemas is partially solvable.

4.5.3.3 The Equivalence Problem

The complement of the equivalence problem is called the non-equivalence problem, i.e., given two compatible Φ-schemas, to decide if the schemas are not equivalent. Similarly, we have the non-generalization problem and the non-inclusion problem.

Lemma 4.8. For schemas

(a) the equivalence problem is not partially solvable,

(b) the non-equivalence problem is not partially solvable,

(c) the generalization problem is not partially solvable,

(d) the non-generalization problem is not partially solvable,

(e) the inclusion problem is not partially solvable,

(f) the non-inclusion problem is not partially solvable.

The parts (c), (d), (e) and (f) follow directly from (a) and (b). Parts (a) and (b) follow from the fact that the equivalence and the non-equivalence problems for one-variable monadic schemas are not partially solvable (see Chapter 3).

4.6 Proofs

4.6.1 Proof of the Translation Lemma

We will only show the following parts of the lemma. The others follow analogously.

(4) $S_1 \leq S_2$ iff $S_1^* \leq S_2^*$.

(7) $C_1 \leq C_2$ iff $C_1^* \leq C_2^*$.

If $S_1 \leq S_2$ then $S_1^* \leq S_2^*$.

Let P denote $\Sigma(S_1^*)$, which is the same as $\Sigma(S_2^*)$, and $S_1 = \langle F_1, \Phi_1, P \rangle$, $S_2 = \langle F_2, \Phi_2, P \rangle$.

Given:

$$\forall I_1 \text{ for } S_1 , \exists I_2 \text{ for } S_2 \text{ s.t. } \exists\theta\text{: } (I_1/P) \leftrightarrow (I_2/P) ,$$
$$\text{if } Val(S_1,I_1) \text{ is defined then } Val(S_2,I_2) = \theta(Val(S_1,I_1)) \quad (a)$$

and

$$\forall I_2 \text{ for } S_2 , \exists I_1 \text{ for } S_1 \text{ s.t. } \exists\theta\text{: } (I_1/P) \leftrightarrow (I_2/P) ,$$
$$\text{if } Val(S_1,I_1) \text{ is defined then } Val(S_2,I_2) = \theta(Val(S_1,I_1)). \quad (b)$$

To prove:

$$\forall I_1^* \text{ for } S_1^* , \exists I_2^* \text{ for } S_2^* \text{ s.t. } \exists\theta\text{: } (I_1^*/P) \leftrightarrow (I_2^*/P) ,$$
$$\text{if } Val(S_1^*,I_1^*) \text{ is defined then } Val(S_2^*,I_2^*) = \theta(Val(S_1^*,I_1^*)) \quad (a')$$

and

$$\forall I_2^* \text{ for } S_2^* , \exists I_1^* \text{ for } S_1^* \text{ s.t. } \exists\theta\text{: } (I_1^*/P) \leftrightarrow (I_2^*/P) ,$$
$$\text{if } Val(S_1^*,I_1^*) \text{ is defined then } Val(S_2^*,I_2^*) = \theta(Val(S_1^*,I_1^*)). \quad (b')$$

We will show (a'), and (b') follows in a similar fashion.

Given any I_1^* for S_1^* , by condition 2(b) of the translation lemma, $\exists I_1$ for S_1 and $\exists\theta_1$: $(I_1^*/P) \leftrightarrow (I_1/P)$ s.t. if $Val(S_1^*,I_1^*)$ is defined then $Val(S_1,I_1) = \theta_1(Val(S_1^*,I_1^*))$. Then, by (a) above, $\exists I_2$ for S_2 and $\exists\theta_2$: $(I_1/P) \leftrightarrow (I_2/P)$ s.t. if $Val(S_1,I_1)$ is defined then $Val(S_2,I_2) = \theta_2(Val(S_1,I_1))$. Finally, by condition 2(a) of the translation lemma, $\exists I_2^*$ for S_2 and $\exists\theta_3$: $(I_2^*/P) \leftrightarrow (I_2/P)$ s.t. if $Val(S_2,I_2)$ is defined then $Val(S_2,I_2) = \theta_3(Val(S_2^*,I_2^*))$. Thus we have a θ $(\theta = \theta_3^{-1} \circ \theta_2 \circ \theta_1)$, θ: $(I_1^*/P) \leftrightarrow (I_2^*/P)$, and if $Val(S_1^*,I_1^*)$ is defined then $Val(S_2^*,I_2^*) = \theta(Val(S_1^*,I_1^*))$.

If $S_1^* \leq S_2^*$ then $S_1 \leq S_2$.

This proof is analogous to the proof above (by interchanging the starred schemas and interpretations with the unstarred ones).

If $\mathcal{C}_1 \leq \mathcal{C}_2$ then $\mathcal{C}_1^* \leq \mathcal{C}_2^*$.

Given: $\forall S_1 \in \mathcal{C}_1 \quad \exists S_2 \in \mathcal{C}_2$ s.t. $S_1 \equiv S_2$. To prove that $\forall S_1^* \in \mathcal{C}_1^* \quad S_2^* \in \mathcal{C}_2^*$ s.t. $S_1^* \equiv S_2^*$.

Notation. If S^* is any conventional schema, and S is the corresponding generalized schema we say $S^* \Rightarrow S$. Also, if S_1, S_2 are any two generalized schemas such that $S_1 = \langle F, \Phi, P_1 \rangle$ and $S_2 = \langle F, \Phi, P_2 \rangle$ and $P_1 \subset P_2$ then, too, we say $S_1 \Rightarrow S_2$.

Proof. Given any $S_1^* \in \mathcal{C}_1^*$. Let $S_1^* \Rightarrow S_1$. Then $S_1 \in \mathcal{C}_1$ by construction of $\mathcal{C}_1$. By hypothesis, $\exists S_2 \in \mathcal{C}_2$ s.t. $S_1 \equiv S_2$. Let $S_1 = \langle F_1, \Phi_1, P \rangle$ where $P = \Sigma(S^*)$ and $S_2 = \langle F_2, \Phi_2, P \rangle$. Now, by the construction of $\mathcal{C}_2$, $\exists S_3 \in \mathcal{C}_2$ and $\exists S_2^* \in \mathcal{C}_2^*$ s.t. $S_2^* \Rightarrow S_3 \Rightarrow S_2$, i.e., $S_3 = \langle F_2, \Phi_2, P_1 \rangle$ and $\Sigma(S_2^*) = P_1 \subset P$. We wish to show that this is the required S_2^*, i.e., $S_1^* \equiv S_2^*$.

Part (i) $S_1^* \underset{\text{gen}}{\leq} S_2^*$.

To prove that $\forall I_1^*$ for S_1^*, $\exists I_2^*$ for S_2^*, P, $\exists \Theta: (I_1^*/P) \rightarrow (I_2^*/P)$, and $\text{Val}(S_2^*, I_2^*) = \Theta(\text{Val}(S_1^*, I_1^*))$ or both are undefined.

For any I_1^* for S_1^*, by 2(b) of the lemma $\exists I_1$ for S_1, $\exists \Theta_1: (I_1^*/P) \rightarrow (I_1/P)$, $\text{Val}(S_1, I_1) = \Theta_1(\text{Val}(S_1^*, I_1^*))$ or both are undefined. Now, as $S_1 \equiv S_2$ we have, by definition, $\exists I_2$ for S_2, $\exists \Theta_2: (I_1/P) \rightarrow (I_2/P)$, and $\text{Val}(S_2, I_2) = \Theta_2(\text{Val}(S_1, I_1))$, or both are

undefined, and as $S_3 \Rightarrow S_2$, $Val(S_3,I_2) = \Theta_2(Val(S_1,I_1))$ or both are undefined. From 2(a), $\exists I_3^*$ for S_2^*, $\exists\Theta_3: (I_3^*/P_1) \leftrightarrow (I_2/P_1)$, $Val(S_2,I_2) = \Theta_3(Val(S_2^*,I_3^*))$. Finally, by 1(b), $\forall I_+^*$ for P, $\exists I_2^*$ for S_2^*,P s.t. $\exists\Theta_4: (I_2^*/P) \leftrightarrow (I_+^*/P)$. We choose I_+^* to be I_3^*. So $\exists I_2^*$ for S_2^*,P, $\exists\Theta_4: (I_2^*/P) \leftrightarrow (I_3^*/P)$, and as $\Sigma(S_2^*) \subset P$ we have $Val(S_2^*,I_3^*) = \Theta_4(Val(S_2^*,I_2^*))$. This gives us the required $\Theta: (I_1^*/P) \leftrightarrow (I_2^*/P)$ s.t. $Val(S_2^*,I_2^*) = \Theta(Val(S_1^*,I_1^*))$, and, in fact, Θ is $\Theta_4^{-1} \circ \Theta_3^{-1} \circ \Theta_2 \circ \Theta_1$.

Part (ii) $S_2^* \underset{gen}{\leq} S_1^*$.

This proof is analogous.

If $\mathcal{C}_1^* \leq \mathcal{C}_2^*$ then $\mathcal{C}_1 \leq \mathcal{C}_2$.

Given $\forall S_1^* \in \mathcal{C}_1^*$ $\exists S_2^* \in \mathcal{C}_2^*$ s.t. $S_1^* \equiv S_2^*$. To prove that $\forall S_1 \in \mathcal{C}_1$ $\exists S_2^* \in \mathcal{C}_2$ s.t. $S_1 \equiv S_2$.

Given any $S_1 \in \mathcal{C}_1$. By the construction of $\mathcal{C}_1$, $\exists S_3 \in \mathcal{C}_1$, $\exists S_1^* \in \mathcal{C}_1^*$ s.t. $S_1^* \Rightarrow S_3 \Rightarrow S_1$. Let $S_1 = \langle F_1,\Phi_1,P_1\rangle$ and $S_3 = \langle F_1,\Phi_1,P\rangle$, $P \subset P_1$. By hypothesis, $\exists S_3^* \in \mathcal{C}_2^*$ s.t. $S_1^* \equiv S_3^*$. Then using the interpolation lemma for conventional schemas (condition 3 of the translation lemma) $\exists S_2^* \in \mathcal{C}_2^*$ s.t. $\Sigma(S_2^*) = \Sigma(S_1) = P$, and $S_2^* \equiv S_1^*$. Let $S_2^* \Rightarrow S_4$, then $S_4 = \langle F_2,\Phi_2,P\rangle$, $S_4 \in \mathcal{C}_2$, and by the construction of $\mathcal{C}_2$, $\langle F_2,\Phi_2,P_1\rangle \in \mathcal{C}_2$. Let S_2 denote $\langle F_2,\Phi_2,P_1\rangle$. This is the desired schema; we have to prove that $S_1 \equiv S_2$.

Part (i) $S_1 \underset{gen}{\leq} S_2$.

To prove that $\forall I_1$ for S_1, $\exists I_2$ for S_2 s.t. $\exists\Theta: (I_1/P) \leftrightarrow (I_2/P)$, and $Val(S_2,I_2) = \Theta(Val(S_1,I_1))$ or both are undefined.

Given any I_1 for S_1. Then I_1 is also for S_2 and $Val(S_1,I_1) = Val(S_3,I_1)$. By 2(a) $\exists I_1^*$ for S_1^*, $\exists\Theta_1: (I_1^*/P) \rightarrow (I_1/P)$ s.t. $Val(S_3,I_1) = \Theta_1(Val(S_1^*,I_1^*))$. As $S_1^* \equiv S_2^*$, we find that $\exists I_2^*$ for S_2^*, $\exists\Theta_2: (I_1^*/P) \rightarrow (I_2^*/P)$ s.t. $Val(S_2^*,I_2^*) = \Theta_2(Val(S_1^*,I_1^*))$. By 2(b), $\exists I_4$ for S_4, $\exists\Theta_3: (I_2^*/P) \rightarrow (I_4/P)$ s.t. $Val(S_4,I_4) = \Theta_3(Val(S_2^*,I_2^*))$ or both are undefined. By 1(a), as I_1 is for P_1 and $(I_4/P) \rightarrow (I_1/P)$, $\exists I_2$ for S_2,P_1, $\exists\Theta_4: (I_2/P_1) \rightarrow (I_1/P_1)$. Hence $\Theta_4^{-1} \circ \Theta_1 \circ \Theta_2^{-1} \circ \Theta_3^{-1}: (I_4/P) \rightarrow (I_2/P)$ and by the well-foundedness of S_4, $Val(S_4,I_2) = \Theta_4^{-1} \circ \Theta_1 \circ \Theta_2^{-1} \circ \Theta_3^{-1}(Val(S_4,I_4)) = \Theta_4^{-1}(Val(S_1,I_1))$ or all diverge. But I_2 is an interpretation for S_2, and $Val(S_2,I_2) = Val(S_4,I_2) = \Theta_4^{-1}(Val(S_1,I_1))$ or all diverge. This completes the proof that $S_1 \underset{gen}{\leq} S_2$.

Part (ii) $S_2 \underset{gen}{\leq} S_1$.

This is proved likewise.

4.6.2 Proof of Theorem 4.1

Given Herbrand schemas $S_1 = \langle F_1,\varphi_1,P\rangle$ and $S_2 = \langle F_2,\varphi_2,P\rangle$, let $\mathcal{H}_1$ be the class of interpretations H for S_1 such that (H/P) is a Herbrand interpretation, and similarly for $\mathcal{H}_2$, then

(a) S_1 halts if and only if it halts on $\mathcal{H}_1$,

(b) S_1 diverges if and only if it diverges on $\mathcal{H}_1$,

(c) $S_1 \equiv S_2$ if and only if $S_1 \equiv S_2$ on $\mathcal{K}_1, \mathcal{K}_2$,

(d) $S_1 \leq S_2$ if and only if $S_1 \leq S_2$ on $\mathcal{K}_1, \mathcal{K}_2$,

(e) $S_1 \underset{gen}{\leq} S_2$ if and only if $S_1 \underset{gen}{\leq} S_2$ on $\mathcal{K}_1, \mathcal{K}_2$,

(f) S_1 is free if and only if it is free on $\mathcal{K}_1$.

Proof: For cases (a), (b), (f) the "only if" part is trivial; and so is the "if" part because if any path is taken by the computation of S_1 on any interpretation I, then the same path is taken by the computation on some interpretation $H \in \mathcal{K}_1$.

We show the theorem for case (e), and the other cases can be proved analogously.

The "only if" part is easy, because given $S_1 \underset{gen}{\leq} S_2$, if $H_1 \in \mathcal{K}_1$ is an interpretation for S_1, then there is an interpretation I_2 for S_2 such that (H_1/P) and (I_2/P) are isomorphic, and the outputs correspond, but there is an interpretation $H_2 \in \mathcal{K}_2$ isomorphic to I_2, and hence we have that (H_1/P) and (H_2/P) are isomorphic, and their outputs correspond.

The "if" part can be proved as follows. Given that $S_1 \underset{gen}{\leq} S_2$ on $\mathcal{K}_1, \mathcal{K}_2$, i.e.,

$\forall H_1$ for S_1, $H_1 \in \mathcal{K}_1$,

$\exists H_2$ for S_2, $H_2 \in \mathcal{K}_2$, and $\mathrm{Val}(S_2,H_2) = \mathrm{Val}(S_1,H_1)$, or both are undefined (note: the isomorphism is identity),

to show that $S_1 \underset{gen}{\leq} S_2$, i.e.,

$\forall I_1$ for S_1

$\exists I_2$ for S_2 , and

$\exists$ an isomorphism $\theta:(I_1/P) \leftrightarrow (I_2/P)$, such that

$Val(S_2,I_2) = \theta(Val(S_1,I_1))$, or both diverge.

Now, given any I_1 for S_1 , by the definition of Herbrand schemas, there exists an H_1 for S_1 such that $(I_1/P) \xrightarrow{h} (H_1/P)$, and $Val(S_1,I_1)$ corresponds to (the term) $Val(S_1,H_1)$, or both diverge. From the hypothesis, there is an H_2 for S_2 , $H_2 \in \mathcal{H}_2$, such that $Val(S_1,H_1) = Val(S_2,H_2)$. And again, as S_2 is a Herbrand schema, for any I_2' for which $I_2' \xrightarrow{h} (H_2/P)$, there is an I_2 for S_2 such that $(I_2'/P) = (I_2/P)$ and $Val(S_2,I_2)$ corresponds to $Val(S_2,H_2)$. We will choose I_2' simply to be (I_1/P) . We now have the desired θ : it is simply the identity function, and either both $Val(S_1,I_1)$ and $Val(S_2,I_2)$ are undefined, or they are equal because both correspond to the same term.

□

4.6.3 Proof of Lemma 4.2

Every well-founded schema is a non-oracle schema.

Given a well-founded schema $S = \langle F,\varphi,P\rangle$ and a path in F from the start statement to a test or a halt statement, we can represent the conjunction of all tests (every test $\alpha(y)$ is changed to $\alpha'()$ by substituting the value of y) executed along this path (or their negations if the false exit is taken by the path) by a formula φ_1 . Then every interpretation that follows this path in the schema satisfies $\varphi \wedge \varphi_1$, and every interpretation that satisfies $\varphi \wedge \varphi_1$ follows this path.

We use the result (see, for example, Shoenfield [1967], Section 5.5, Lemma 4) that given sentences η, ψ', and a set P of functions and predicates, if whenever $I_1 \vdash \eta$, $I_2 \vdash \eta$, and (I_1/P) isomorphic to (I_2/P) we have $I_1 \vdash \psi'$ if and only if $I_2 \vdash \psi'$, then there exists a quantifier free sentence ψ over P such that $\eta \rightarrow (\psi' \leftrightarrow \psi)$ is valid.

Suppose our given path in F leads to a test statement, then the test can be represented as a simple atomic test α only on constant terms, and we have, by the well-foundedness of S, that whenever $I_1 \vdash \varphi \wedge \varphi_1$, $I_2 \vdash \varphi \wedge \varphi_1$, (I_1/P) isomorphic to (I_2/P) we have $I_1 \vdash \alpha$ if and only if $I_2 \vdash \alpha$. We hence have a sentence ψ such that $\varphi \wedge \varphi_1 \rightarrow (\alpha \leftrightarrow \psi)$, and by the deduction theorem $\varphi \wedge \varphi_1 \vdash (\alpha \leftrightarrow \psi)$, i.e., for all interpretations that follow this path, the outcome of the test equals the value of the quantifier free formula ψ over P (which is the requirement for a non-oracle schema).

If, on the other hand, the given path in F leads to a halt statement, then the output is some (constant) term $\tau()$. If we now introduce a new zero-ary function a_0 into interpretations for the schema, we have that whenever $I_1 \vdash \varphi \wedge \varphi_1$, $I_2 \vdash \varphi \wedge \varphi_1$, $(I_1 / P \cup \{a_0\})$ isomorphic to $(I_2 / P \cup \{a_0\})$, we have $I_1 \vdash a_0 = \tau()$ if and only if $I_2 \vdash a_0 = \tau()$ by the well-foundedness of S, and hence there is a formula $\psi(a_0)$ (we call it $\psi(a_0)$ instead of ψ for convenience) such that $\varphi \wedge \varphi_1 \rightarrow (a_0 = \tau() \leftrightarrow \psi(a_0))$. But a_0 doesn't appear in $\varphi \wedge \varphi_1$, and hence $\varphi \wedge \varphi_1 \rightarrow \forall x(x = \tau() \leftrightarrow \psi(x))$, and again by the deduction theorem $\varphi \wedge \varphi_1 \vdash \forall x(x = \tau() \leftrightarrow \psi(x))$, which is the desired result.

This concludes the proof of Lemma 4.2. □

4.6.4 Proof of Theorem 4.3

The class $\mathcal{C}$ of uninterpreted schemas is equivalent to the class $\mathcal{C}(A,=)$ of φ-schemas corresponding to the conventional schemas with arrays and equality tests.

Given a schema $S = \langle F,\varphi,P\rangle$ in $\mathcal{C}$ we will construct a conventional schema S^* with arrays (and counters) and equality tests having the symbols of P as its base functions and predicates, such that for any interpretation I for S, $Val(S,I) = Val(S^*,I/P)$. We can then translate S^* into a generalized schema S_1 in the standard way (see Section 4.4). It should be noted that since it is unsolvable if any given schema S is an element of $\mathcal{C}$, our translation process will go through even for schemas not in $\mathcal{C}$. However, it will not necessarily be correct. If the given schema S is interpreted, then S_1 will not be equivalent to S, but will be a strict generalization. If S is not well founded, then, of course, equivalence is not well defined.

We will make use of the fact that a conventional schema with counters can simulate the behavior of any schema except when it comes to making tests, or halting, in which case, it has to make use of its base functions and predicates.

S^* proceeds as follows. It simulates the computation of S, keeping track of the value of the single variable of S (as a constant term). It also keeps track of any tests that S has made along the path. This is kept as a formula $\alpha = \alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n$ where each α_i is an atomic formula or a negated atomic formula. When S comes to a test β, S^* enumerates all valid formulas until it comes to one of the form

$$\varphi \wedge \alpha \rightarrow (\psi \leftrightarrow \beta)$$

where ψ is a quantifier free formula that uses only the base functions and predicates from P (note: we are using here the completeness theorem for first order predicate calculus with equality, and the fact that S is non-oracle). S^* then makes the appropriate tests to determine if ψ is true or false, and updates α to $\alpha_1 \wedge \ldots \wedge \alpha_n \wedge \beta$ if ψ is true, or to $\alpha_1 \wedge \ldots \wedge \alpha_n \wedge \neg\beta$ if ψ is false. When S comes to a halt statement HALT($\tau(y)$), S^* enumerates all valid formulas until it comes to one of the form

$$\varphi \wedge \alpha \rightarrow \forall x((x = \tau) \leftrightarrow \psi(x))$$

where τ represents $\tau(y)$ in which the value of y (as a term) is substituted for y, and $\psi(x)$ is quantifier free, and uses only the symbols of P. When such a formula is found, S^* enumerates all elements reachable by applying functions of P, and halts on the first element x for which $\psi(x)$ is true.

A final note seems to be in order. To be very formal, the class $\mathcal{C}(A,=)$ is to be interpreted not just as the class $\mathcal{C}$ of schemas corresponding to the conventional schemas with arrays and equality, but the class obtained by renaming the function and predicate symbols of schemas (in $\mathcal{C}$) in all possible ways (distinct symbols must, of course, remain distinct). The reason is that in the translating process we used certain function and predicate symbols which couldn't appear in the set P of any schema $\langle F,\varphi,P\rangle$ in $\mathcal{C}$.

□

4.6.5 Proof of Theorem 4.4

Every schema in the class of uninterpreted Herbrand schemas can be effectively translated into an equivalent Φ-schema corresponding to a conventional schema with arrays.

Given an uninterpreted Herbrand schema $S = \langle F,\Phi,P\rangle$ we construct conventional schema S^* with arrays (and counters), as in the previous section, such that the generalized schema corresponding to S^* is equivalent to S.

S^* simulates the computation of S, keeping track of the value of the single variable of S (as a constant term). It also keeps track of the tests S has made along the path of the computation, as a formula α. When S comes to a test β, S^* enumerates all valid formulas until it comes to one of the form

$$\varphi \wedge \alpha \rightarrow (\psi \leftrightarrow \beta)$$

where ψ is quantifier-free and is over P. (Actually we can show that there always exists an equality-free ψ of this kind, but that is unnecessary.) S^* now makes the appropriate tests to determine ψ for Herbrand interpretations. For this reason it doesn't need to make any tests of equality. The same exit would be taken for all interpretations for S by the Herbrand property, and hence S^* can update α and continue simulation of S.

When S comes to a halt statement $HALT(\tau(y))$, S^* enumerates all valid formulas until it finds one of the form

$$\varphi \wedge \alpha \rightarrow \forall x((x = \tau) \leftrightarrow \psi(x))$$

where τ represents $\tau(y)$ with the value of y substituted for the variable y; and $\psi(x)$ is quantifier-free, and over P. S^* enumerates

all elements reachable by applying functions of P, and halts on the first element x for which $\psi(x)$ is true assuming a Herbrand interpretation. When S^* is converted to a generalized schema S_1, the outputs of S and S_1 are the same for all interpretations by the Herbrand property of S.

□

4.6.6 Proof of Theorem 4.5

To show that the halting problem for Φ-schemas is partially solvable.

The partial solvability of the halting problem can be shown by reducing this problem to the validity problem of formulas of first order predicate calculus, with equality, which is partially solvable. We use the approach used by Manna [1968, 1969]. Given a flowchart F, we associate with F formula $\psi(F)$ of predicate calculus such that F halts for all interpretations if andonly if $\psi(F)$ is valid, $\psi(F)$ is constructed as follows. Let all statements of F be labeled $L_1,\ldots,L_n$. Associate, with each statement L_i, a predicate q_i. Let ψ' be the conjunction of the axioms obtained as shown below:

Statement	Axiom
START $y \leftarrow \tau()$; goto L_i	$q_i(\tau())$
L_i: HALT($\tau(y)$)	$\forall x\ q_i(x) \rightarrow q$
L_i: LOOP	(no axiom)
L_i: $y \leftarrow \tau(y)$; goto L_j	$\forall x\ q_i(x) \rightarrow q_j(\tau(x))$
L_i: if $\alpha(y)$ then goto L_j else goto L_k	$\forall x\ q_i(x) \wedge \alpha(x) \rightarrow q_j(x)$ $\wedge\ q_i(x) \wedge \neg\alpha(x) \rightarrow q_k(x)$

Then $\psi(F)$ is $\psi' \rightarrow q$ (q is introduced in the axiom for a halt statement). We then find the schema $\langle F,\varphi,P\rangle$ halts if and only if $\varphi \rightarrow \psi(F)$ is valid.

□

References

Ashcroft, Manna and Pnueli [1970]. E. Ashcroft, Z. Manna and A. Pnueli, "Decidable properties of monadic functional schemas," in Theory of Machines and Computations, (Kohavi and Paz, Eds.), Academic Press, pp. 3-18.

Chandra [1972]. A. K. Chandra, "Efficient compilation of linear recursive programs," Report no. CS-282, Computer Science Dept., Stanford University, April 1972.

Chandra and Manna [1970]. A. K. Chandra and Z. Manna, "Program schemas with equality," in Proceedings of the Fourth Annual ACM Symposium on the Theory of Computing, Denver, Colorado, May 1-3, 1972, pp. 52-64.

Cadiou [1972]. J. M. Cadiou, "Recursive definitions and their computations," Ph.D. Thesis, Report no. CS-266, Computer Science Dept., Stanford University, March 1972.

Constable and Gries [1972]. R. L. Constable and D. Gries, "On classes of program schemata," in SIAM Journal of Computing, Vol. 1, No. 1, March 1972, pp. 66-118.

Garland and Luckham [1971]. S. J. Garland and D. C. Luckham, "Program schemes, recursion schemes, and formal languages," UCLA report no. ENG-7154, June 1971.

Hewitt [1970]. C. Hewitt, "More comparative schematology," Artificial Intelligence Memo, no. 207, Project MAC, Mass. Institute of Technology, August 1970.

Hopcroft and Ullman [1969]. J. E. Hopcroft and J. D. Ullman, Formal Languages and Their Relation to Automata, Addison-Wesley, 1969.

Ianov [1958]. Iu Ianov, "The logical schemas of algorithms," Problems and Cybernetics, Vol. 1, pp. 75-127, (Russian edition).

Ianov [1960]. Iu Ianov, "The logical schemes of algorithms," English translation in Problems of Cybernetics, Vol. 1, Pergamon Press, New York, 1960, pp. 82-140.

Karp and Miller [1969]. R. M. Karp and R. E. Miller, "Parallel program schemata," Journal of Computer and System Sciences, Vol. 3, No. 2, May 1969, pp. 147-195.

Luckham, Park and Paterson [1970]. D. C. Luckham, D. M. R. Park and M. S. Paterson, "On formalized computer Programs," Journal of Computer and System Sciences, Vol. 4, No. 3, (June 1970), pp. 220-249.

Manna [1968]. Z. Manna, "Termination of algorithms," Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburgh, Pa., April 1968.

Manna [1969]. Z. Manna, "Properties of programs and the first-order predicate calculus," Journal of the ACM, Vol. 16, No. 2, April 1969, pp. 244-255.

McCarthy [1962]. J. McCarthy, "Towards a mathematical science of computation," PROC. IFIP, 1962, pp. 21-34.

McCarthy [1963]. J. McCarthy, "A basis for a mathematical theory of computation," from Computer Programming and Formal Systems, North-Holland, Amsterdam, 1963, pp. 33-70.

Miller [1972]. R. E. Miller, "A boundary between decidability and undecidability for parallel program schemata," in Proceedings of an ACM Conference on Proving Assertions about Programs, Las Cruces, New Mexico, Jan. 6-7, 1972, pp. 116-120.

Milner [1970]. R. Milner, "Equivalences on program schemas," Journal of Computer and System Sciences, Vol. 4, No. 3, June 1970, pp. 205-219.

Morris [1968]. J. H. Morris, "Lambda-calculus models of programming languages," Ph.D. Thesis, Project MAC, Mass. Institute of Technology, MAC-TR-57, (December 1968).

Morris [1972]. J. H. Morris, "Recursion schemas with lists," in Proceedings of the Fourth Annual ACM Symposium on the Theory of Computing, Denver, Colorado, May 1972, pp. 35-44.

Paterson [1967]. M. S. Paterson, "Equivalence problems in a model of computation," Ph.D. Thesis, University of Cambridge, England (August 1967). Also Artificial Intelligence Memo, No. 1, Mass. Institute of Technology, 1970.

Paterson [1968]. M. S. Paterson, "Program schemata," in Machine Intelligence 3 (Michie, Ed.), Edinburgh University Press, pp. 19-31.

Paterson and Hewitt [1970]. M. S. Paterson and C. E. Hewitt, "Comparative schematology," in Record of Project MAC Conference on concurrent systems and parallel computation, ACM, New York (December 1970), pp. 119-128.

Plaisted [1972]. D. Plaisted, "Program schemas with counters," Proceedings of the Fourth Annual ACM Symposium on the Theory of Computing, Denver, Colorado, May 1-3, 1972, pp. 44-51.

Rogers [1967]. H. Rogers, Theory of Recursive Functions and Effective Computability, McGraw-Hill, 1967.

Rutledge [1964]. J. D. Rutledge, "On Ianov's program schemata," J.ACM, Vol. 11, No. 1, (January 1964), pp. 1-9.

Shoenfield [1967]. J. R. Shoenfield, Mathematical Logic, Addison-Wesley (1967).

Strong [1971a]. H. R. Strong, "Translating recursion equations into flowcharts," Journal of Computer and System Sciences, Vol. 5 (June 1971), pp. 254-285.

Strong [1971b]. H. R. Strong, "High level languages of maximum power," Proc. IEEE Conference on Switching and Automata Theory, 1971, pp. 1-4.